1975

A formal analysis of name accessing in programming languages

Carol Lynn Smith
Iowa State University

A formal analysis of name accessing in programming languages

by

Carol Lynn Smith

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1975

TABLE OF CONTENTS

	Page
CHAPTER I. INTRODUCTION	1
Significance of the Name Accessing Issue	1
Outline of the Dissertation	9
CHAPTER II. HISTORICAL PERSPECTIVES	12
Motivation for Semantic Modeling	12
Mathematical Models	16
Axiomatic Models	20
Operational Models	21
Semantics of Data Structures	25
Semantics of Name Accessing	27
Applicability of Formal Semantic Models	30
CHAPTER III. THE ACCESSING GRAPH MODEL	33
Basic Name Accessing Concepts	33
Primitive Notions About Naming	34
Description of the Accessing Graph Model (AGM)	41
CHAPTER IV. APPLICATIONS OF THE MODELING TECHNIQUE	50
Mini-Language 1 (ML-1)	50
Mini-Language 2 (ML-2)	62
Mini-Language 3 (ML-3)	76
Mini-Language 4 (ML-4)	96
CHAPTER V. NAME ACCESSING IN THE SYMBOL-2R PROGRAMMING LANGUAGE	119
CHAPTER VI. CONCLUSIONS AND FUTURE RESEARCH	154
BIBLIOGRAPHY	158
ACKNOWLEDGMENTS	165

LIST OF FIGURES

Page

Figure 1. Execution trace for program FIG1 8

Figure 2. Program node ≈ program node relationships. 43

Figure 3. Program node ≈ data node relationships 46

Figure 4. Data node ≈ data node relationships. 47

Figure 5. Syntactic description of mini-language 1 51

Figure 6a. Initial AGM representation for example 5 with state = (G_0, σ) 54

Figure 6b. Execution trace for example 5 with state = (G_1, σ^2) 55

Figure 6c. Execution trace for example 5 with state = (G_2, σ^3) 56

Figure 6d. Execution trace for example 5 with state = (G_3, σ^4) 57

Figure 6e. Execution trace for example 5 with state = (G_4, σ^5) 58

Figure 6f. Execution trace for example 5 with state = (G_5, σ^6) 59

Figure 6g. Execution trace for example 5 with state = (G_6, σ^7) 60

Figure 6h. Execution trace for example 5 with state = (G_7, σ^8) 61

Figure 7. Syntactic description of mini-language 2 62

Figure 8a. Initial AGM representation for example 6 with state = (G_0, σ) 68

Figure 8b. Execution trace for example 6 with state = (G_6, σ^7) 69

Figure 8c. Execution trace for example 6 with state = (G_9, σ^{10}). 70

Page

Figure 8d. Execution trace for example 6 with state = (G_{10}, σ^{11}) 71

Figure 8e. Execution trace for example 6 with state = (G_{13}, σ^{14}) 72

Figure 8f. Execution trace for example 6 with state = (G_{16}, σ^{17}) 73

Figure 9. Syntactic description of mini-language 3 77

Figure 10a. Initial AGM representation for example 7 with state = (G_0, σ) 87

Figure 10b. Execution trace for example 7 with state = (G_6, σ^7) 88

Figure 10c. Execution trace for example 7 with state = (G_{11}, σ^{12}) 89

Figure 10d. Execution trace for example 7 with state = (G_{12}, σ^{17}) 90

Figure 10e. Execution trace for example 7 with state = (G_{20}, σ^{21}) 91

Figure 11. Syntactic description of mini-language 4 97

Figure 12a. Initial AGM representation for example 8 with state = (G_0, σ) 110

Figure 12b. Execution trace for example 8 with state = $(G_5, \sigma n^6)$. 111

Figure 12c. Execution trace for example 8 with state = $(G_9, \sigma n^{10})$ 112

Figure 12d. Execution trace for example 8 with state = $(G_{14}, \sigma n^{10} \sigma B \sigma n^4)$ 113

Figure 13. Execution trace for example 9 with state = $(G_{16}, \sigma n^{11} \sigma B \sigma n^2 \sigma B \sigma n^2)$. 118

Figure 14. Explicit declaration of identifiers. 120

Figure 15. Retention and the nondecrementing clock 122

Page

Figure 16a. Initial AGM representation for example 10 with state = $(G_0, \sigma n)$ 124

Figure 16b. Execution trace for example 10 with state = $(G_0, \sigma n^3)$ 125

Figure 16c. Execution trace for example 10 with state = $(G_1, \sigma n^4)$ 126

Figure 16d. Execution trace for example 10 with state = $(G_3, \sigma n^6)$ 127

Figure 16e. Execution trace for example 10 with state = $(G_3, \sigma n^7)$ 128

Figure 17a. Initial AGM representation for example 11 with state = $(G_0, \sigma n)$ 131

Figure 17b. Execution trace for example 11 with state = $(G_2, \sigma n^5)$ 132

Figure 17c. Execution trace for example 11 with state = $(G_4, \sigma n^{10})$ 133

Figure 17d. Execution trace for example 11 with state = $(G_5, \sigma n^{13})$ 134

Figure 18a. Initial AGM representation for example 12 with state = $(G_0, \sigma n)$ 136

Figure 18b. Execution trace for example 12 with state = $(G_2, \sigma n^8)$ 137

Figure 18c. Execution trace for example 12 with state = $(G_5, \sigma n^8 \sigma B \sigma n^4)$ 138

Figure 18d. Execution trace for example 12 with state = $(G_6, \sigma n^9)$ 139

Figure 19a. Initial AGM representation for example 13 with state = $(G_0, \sigma n)$ 141

Figure 19b. Execution trace for example 13 with state = $(G_4, \sigma n^3 \sigma B \sigma n^4)$ 142

Figure 19c. Execution trace for example 13 with state = $(G_6, \sigma n^5)$ 143

Page

Figure 20a. Initial AGM representation for example 14 with state = (G_0, σ) 145

Figure 20b. Execution trace for example 14 with state = (G_4, $\sigma n^6 \sigma B \sigma n$) 146

Figure 21a. Initial AGM representation for example 15 with state = (G_0, σn) 150

Figure 21b. Execution trace for example 15 with state = (G_4, σn^{10}) 151

Figure 22. Link simulation of a nonparameterized procedure . . . 152

Figure 23. Link simulation of a parameterized procedure. 153

LIST OF DIAGRAMS

Page

Diagram 1. Preservation of share closures. 102

Diagram 2. Translation of a procedure text 104

Diagram 3. Share and value closures 106

CHAPTER I. INTRODUCTION

Significance of the Name Accessing Issue

One of the main purposes of a programming language is to provide a framework within which a user can convey the specification of a task to an information processing system. The user wants to process some information or, more specifically, a set of information structures. In this regard, one of the most important functions performed by the programming language is the naming of these information structures. Using a programming language, the user can name these structures and then convey the naming conventions to the information processing system. The system must decipher (and adhere to) the naming conventions and process the named structures as specified in the program.

Most, if not all, programming languages provide mechanisms for users to introduce and use names. Although these naming mechanisms vary widely from language to language, the ultimate intent is still the same: the user is provided with naming conventions for naming information and then subsequently referencing this named information. The naming conventions of a given language determine precisely how information can be named and referenced using that language. In referencing named information, it is necessary to determine the meaning of a name at some point in the computation, the point of reference. The determination of the meaning of a name will be referred to as the name accessing problem.

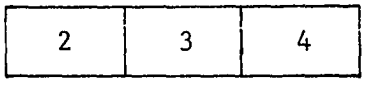
Unfortunately, the establishment of naming conventions for a

programming language and the subsequent solution to the name accessing problem has not proven to be an easy task. From a designer's point of view the more generality or flexibility allowed by the naming conventions the more expressive power the language has. On the other hand, the more generality allowed by the naming conventions the more difficult and expensive the implementation and/or semantic definition of the conventions. Indeed, the subtleties introduced by these often conflicting points of view have frequently confused both designers and implementors (not to mention users). The following paragraphs outline two examples of these subtleties.

In ALGOL 60, the concept of an own variable was introduced to provide a mechanism for communication between different instances of a block's execution. Storage for an own variable is allocated upon the first entry to the block or procedure in which it is declared. This storage persists, even when the block or procedure is exited, and becomes accessible upon a subsequent entry. One problem that developed with the use of own variables is that of naming conventions for own arrays. In ALGOL 60, the subscript bounds for arrays are specified at block entry and these bounds may vary for different entries to a block. Consider, for example, the following ALGOL 60 program.

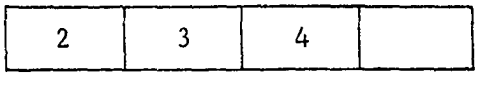
```
begin integer x;
      procedure f(n); value n; integer n;
        begin own array A[n:n*2];
              integer i;
          if x = 0 then
              for i:=n step 1 until n*2 do A[i]:=i;
          if x = 1 then x:=A[n]
        end f;
  x:=0;
  f(2);
  x:=1;
  f(3)
end
```

The first call on f creates storage for the array A consisting of 3 elements. Values are assigned to these elements as depicted below.



A[2] A[3] A[4]

The problem arises when the second call to f is made and A is declared to be an array consisting of four elements. The storage for A, allocated in the first call, already exists and the assigned values have been retained. The question is, how does this old storage relate to the newly declared array. If we associate the base of the old array with the base of the new array we would obtain the following overlay conditions.

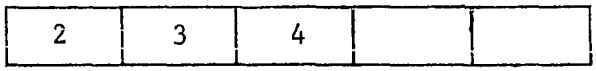


old	A[2]	A[3]	A[4]		
new	A[3]	A[4]	A[5]	A[6]	

Under this interpretation A[6] would be initially undefined since A was expanded from a 3 element array to a 4 element array. Execution of the

line x:=A[n] would result in x obtaining the value "2".

Resolution on the basis of base elements, however, is not used as the defining rule in the Revised ALGOL 60 report (54). The report specifies that the relationship between the old and new arrays be resolved on the basis of names. For example, if A[3] was a legal reference to a component of A in the old array and is also a legal reference to a component in the new array, then the new A[3] refers to the storage allocated for the old A[3]. Using this scheme we would then obtain the following:



old	A[2]	A[3]	A[4]		
new		A[3]	A[4]	A[5]	A[6]

Note that now A[5] and A[6] are initially undefined since these are names that did not exist in the first execution of f. Furthermore, execution of the line x:=A[n] would result in x obtaining the value 3.

The crucial aspect of this type of resolution is the status of the old element A[2]. Even though this storage has been saved and the value "2" retained, it has now become inaccessible since A[2] is not a legal name within the context of the most recently calculated subscript bounds for A. In order to eventually reclaim this location some type of shifting operation must be defined. A shifting algorithm defined for this purpose can be found in Ingerman (34). The overhead associated with such a shifting operation has dampened the spirits of several implementors and dynamic own arrays have generally been excluded from

most implementations including the IFIP ALGOL subset (33).

The difficulty with dynamic own arrays can be attributed to a designer's lack of foresight into the ramifications of combining the own concept with dynamic arrays. As an example of an implementor's failure to correctly decipher and adhere to the naming conventions specified in the language design, let us consider the "most-recent" error uncovered in the PL/I F compiler (27).

In dealing with a block structured language, such as PL/I (or ALGOL 60), storage is normally allocated upon block entry and deallocated upon subsequent block exit. The storage allocation routines for these languages tend to simulate the actions of a pushdown stack in that storage is created and destroyed on a last-in/first-out basis.

One of the naming problems that must be considered in block structured languages is exactly how nonlocal variables (variables used in but not declared in a block) are resolved. In the ALGOL-like program below, the name A is nonlocal to the procedure F. The correct resolution of A does not depend on the last-in/first-out storage model supported by both PL/I and ALGOL. Instead the following rule is used for these cases: An identifier which is nonlocal to a block or procedure x is said to have the same meaning as it does in the block which lexicographically (i.e., statically) encloses x.

```
begin integer A;
      procedure F;
         print (A)
      end F;
   A:=1;
   begin integer A;
         A:=2;
         F
   end
end
```

Using this rule would result in the resolution of the nonlocal A in F to the A which is declared in the outer block. Execution of the print statement would therefore result in the output of the value "1".

Since the resolution of names is not related to storage allocation, chains or pointers are usually used to resolve nonlocals. If we define the storage allocated for a block x to be an activation record for the block x, then we could associate with this activation record a pointer to the activation record for the lexicographically enclosing block. This pointer could then be used to resolve the nonlocals of x. A point of confusion arises, however, when more than one activation record exists for the lexicographically enclosing block. This situation would occur with recursion for instance. It is precisely this point which led to the so-called "most-recent" error.

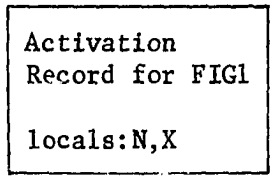
In a paper discussing recursion in ALGOL 60, Dijkstra (13) addressed this issue. He postulated (incorrectly) that the pointer corresponding to a procedure activation should point to "the most recent, not yet completed, activation of the first block that lexicographically encloses the block of the subroutine called in." This interpretation works in many cases but fails when either procedures or labels are

allowed to be passed as parameters. The PL/I program below illustrates the error for procedure parameters. The reader is referred to (50) for a treatment of label parameters and a complete discussion of the "most-recent" error.

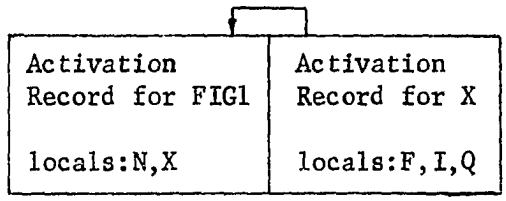
```
FIG1:  PROCEDURE OPTIONS (MAIN);
       DECLARE N FIXED INIT(3);
      X:  PROCEDURE(F) RECURSIVE;
          DECLARE F ENTRY;
          DECLARE I FIXED INIT(0);
             Q:PROCEDURE;
                I = I + 1;
                END Q;
          N = N - 1;
          IF N > 0 THEN CALL X(Q);
                   ELSE CALL F;
          PUT LIST (I);
         END X;
      CALL X(X);
END FIG1;
```

A trace of the program FIG1 is given in Figure 1. The crucial step occurs when the call on F is executed, i.e., in going from (d) to (e) in Figure 1. According to Dijkstra and the original PL/I F compiler, the pointer associated with the activation record for F (or more precisely Q) would point to the most recent activation record for X as shown by the dotted arrow. However, since the formal parameter F is actually bound to the Q declared in the second call on X, the pointer should point to the activation record associated with the second call on X. This pointer is indicated by the solid arrow. A correct implementation of the naming convention would result in the output of "0,1,0". In other words, the call on Q would update the I declared local to the second call on X.

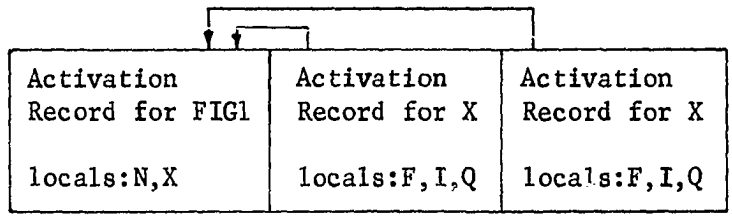
The difficulty with dynamic own arrays and the "most-recent" error



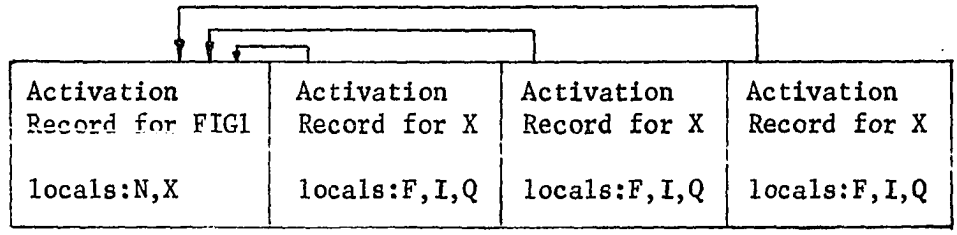
(a) prior to first call on X (N=3)



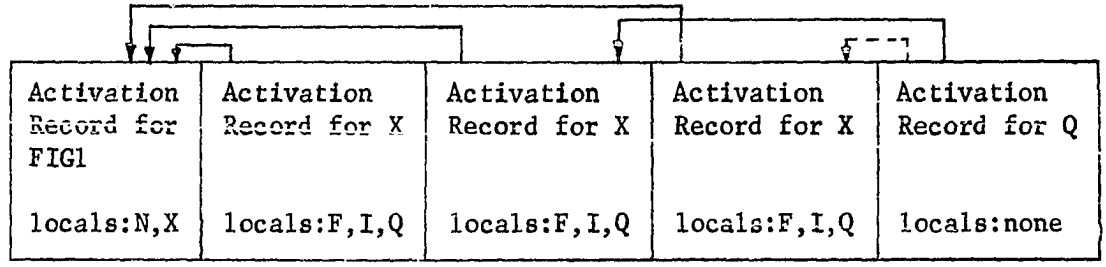
(b) prior to second call on X (N=2)



(c) prior to third call on X (N=1)



(d) prior to call on F (N=0)



(e) inside Q

Figure 1. Execution trace for program FIG1

exemplify the fact that certain naming conventions can create non-transparent name accessing problems. In most cases, some type of formal semantic model is needed to help discover and hopefully solve these problems. It has been recognized for some time that such a model is badly needed (28, 55, 75), but unfortunately few such models exist. It is not intended that the model presented in this dissertation will constitute a general theory of names. It merely represents a step in that direction. Within the remaining chapters, it is argued that this model not only encompasses a wide variety of name accessing mechanisms, but also is applicable in the modeling of the name accessing capabilities of most existing programming languages.

The model presented here is an operational model in the sense that it represents computations as sequences of transformations upon information structures. The transformations are effected by an abstract computer whose instruction set is tailored specifically to name accessing issues. The structure of this abstract computer and the underlying information structures upon which it operates constitute the model.

Outline of the Dissertation

Since a great deal of work has been undertaken in the area of semantic modeling of computational processes, it is appropriate to review this work before presenting the new model. The first part of Chapter II is therefore devoted to a discussion of various semantic models and their relevance in modeling name accessing issues. The second part of the chapter then discusses two approaches to naming due to

Gilmore (24) and Boyle and Grau (6) respectively. The strengths, weaknesses and scope of each of the two approaches are discussed briefly.

The naming model itself is presented in Chapter III. First, a discussion is presented concerning what names are, how names are represented in programs, and what mechanisms typically are available (in programming languages) for the manipulation of names. A set of primitives is then developed to represent a basic set of operations upon names and also to define precisely what is meant by a name in the abstract world of a computation. A special metalanguage, the Name Accessing Language (NAL), is presented to provide a common framework for analyzing and comparing the name accessing properties of programming languages. The notion of accessing graphs is then developed as a descriptive device for name accessing. This notion leads to an Accessing Graph Model, called the AGM, in which the accessing of a name is equated with the traversal of a path in an accessing graph. Finally, a simple abstract computer is presented which interpretively executes, in NAL code, programs represented as accessing graphs.

In Chapter IV, NAL and the AGM are tested for their ability to model some name accessing mechanisms typically found in programming languages. Several mini-languages are presented which highlight various naming conventions. These include

1) a nonblock structured language,
2) ALGOL-like block structure,
3) a non-ALGOL block structure, and
4) procedure closures.

For each mini-language, a method is outlined indicating how the name accessing mechanisms of the language are translated into NAL primitives and subsequently defined in terms of the AGM. Examples are provided to illustrate the execution of programs written in the mini-languages.

Having demonstrated in Chapter IV that NAL and the AGM are sufficiently general to model a large class of name accessing features, Chapter V continues with an examination of the usefulness of this modeling technique in handling name accessing issues arising in existing programming languages. In particular, this approach is used in modeling an actual programming language, the SYMBOL-2R Programming Language (56). This language was chosen for this role because, in addition to containing numerous name accessing features available in most high-level languages, it also supports a wide variety of binding times (the times at which information is associated with names).

Chapter VI concludes this dissertation by summarizing the conclusions of this work and by suggesting possible extensions and avenues of future research.

CHAPTER II. HISTORICAL PERSPECTIVES

Motivation for Semantic Modeling

The purpose of this section is to provide a historical account of research in programming language semantics. It is difficult to state exactly when and why this research was initiated but it seems quite likely that the publication of the Revised ALGOL 60 report (54) stimulated much of this work. The contrast between the precise, clear BNF syntactic description of ALGOL 60 and the imprecise and often obscure natural language semantic description focused attention on the need for a formal semantic descriptive model. It was hoped that a semantic model could be developed that would contribute as much to semantic definition as BNF had contributed to syntactic description.

Prior to 1962, very little attention had been paid to semantic descriptions. About this time, Steel summarized the state of the art in this area as follows: ". . . the only existing method that adequately describes a programming language is the exhibition of a machine language version of a compiler for the programming language being described, together with a citation of the explicit machine on which the compiler is expected to function" (66, p. 25). Over the past 12 years, a great deal of research has been directed toward improving this situation. Before we can judge how successful these attempts have been, it will be necessary to consider the exact objectives of this research.

The motivation for a formal semantic definition is best understood

in terms of the needs of those groups who would use such a definition. Historically, it has been claimed that a formal semantic definition should be useful to language designers, language implementors and users (3). In this survey another group will be considered; namely the theoretical computer scientist who is interested in studying programming languages and programming language semantics as objects in their own right. The criteria which a formal semantic model should satisfy in order to be useful to these groups are outlined below.

I. Language designers

   A. The model should provide a theoretical framework for the design and comparison of various language constructs.

   B. The model should allow for the detection of ambiguous, incompatible or contradictory language constructs. Ideally, it should also act as a catalyst in the design of new language constructs.

   C. The model should allow small changes in the language constructs to be realized by small changes in the language description.

   D. The model should describe independent language constructs independently.

II. Language implementors

   A. The model should clearly indicate to the implementor those parts of the language definition that are incompletely specified and therefore left to the implementor to define.

The model must also indicate the restrictions that must be adhered to by the implementor in completing the definition.

B. The model should clearly delimit the actions which can be performed at compile-time as opposed to those which must be done at run-time.

C. The model should provide a framework within which the implementor can verify that the implementation methods he had chosen satisfy the language definition. Furthermore, the model should allow for the comparison of various meaning-preserving implementations.

D. In general, a formal semantic model should provide the implementor with a complete, precise and unambiguous definition of the language he wishes to implement.

III. Language user

A. The formal model should provide the user with a concise, readable and transparent description of the language.

B. The model should provide a framework conducive to studying and proving properties about individual programs.

IV. Theoretical computer scientist

A. The model should be applicable to a large class of programming languages. This would enable the computer scientist to use a common framework in which to analyze and compare various languages.

The criteria which have been listed are quite demanding. Hoare and

Lauer (30) have recently suggested that more than one model may be necessary to satisfy the needs of these diverse groups. In any event, it seems appropriate at this time (after many years of research) to see to what extent these objectives have been met and the formalisms that have been used in this endeavor.

To the best of the author's knowledge, no thorough attempt has been made to systematically classify and analyze the research that has been undertaken in specifying the semantics of programming languages. It is not intended that this task will be accomplished in this section, but steps in this direction will be presented.

For purposes of comparison, three descriptive categories will be used to distinguish semantic approaches: mathematical approaches, axiomatic approaches and operational approaches. These categories are derived from Wegner's analysis of semantic modeling techniques (76). Mathematical semantic models treat the objects they represent as if they have an existence independent of any particular representation. In this approach, a mathematical formalism is used as a host for the programming language constructs which are being defined. The semantics of the language constructs are then defined in terms of the semantics of the mathematical host. The axiomatic approach is based on the foundations of mathematical logic to make assertions about the state of a computing system at various points of program execution. The semantics of a programming language construct is defined in terms of the assertions that can be made prior to execution of the construct (input assertions) and the assertions that can be made after its execution (output assertions). The operational

approach comes closest to modeling the actual execution of a computing device. If the mathematical and axiomatic approaches are characterized by the fact that they are concerned with what is computed, then it would be appropriate to state that in the operational approach emphasis is placed on how the computations are performed. In the operational approach, information structures are used to model the computer state. The semantics of a language construct is then defined in terms of the effect its execution has on the information structures.

Numerous semantic models have been proposed using the semantic approaches described above. In the following paragraphs, some of these attempts will be outlined.

Mathematical Models

Several semantic models have been proposed which use the mathematical formalism of λ-calculus notation (11). This survey will consider two of these, the models proposed by Landin (40, 41) and Strachey (67).

The popularity of the λ-calculus is related to the fact that there is some degree of similarity between λ-calculus constructs and certain programming language constructs. The obvious correspondence between the notions of local and nonlocal variables in programming languages and the concepts of free and bound variables in the λ-calculus provides a natural mechanism for modeling ALGOL-like block structure. In addition, the functional nature of the λ-calculus provides a natural model for function definition and function application. Other programming language constructs, such as assignment statements and jumps, are not easily modeled

in the λ-calculus. The semantic models proposed by Landin and Strachey have used different approaches in overcoming these drawbacks.

Landin has proposed the definition of programming language constructs in terms of their "compilation" into extended λ-calculus expressions. The extensions, program points and assigners, are included to handle labels and assignments. Landin's approach can be outlined as follows: 1) translate the language constructs that are being defined into extended λ-expressions, 2) specify a mechanical evaluation procedure for the extended λ-expressions, and 3) identify the semantics of a language construct with its associated evaluation in extended λ-calculus form.

The pure λ-calculus already has the notion of evaluation specified in the formalism by means of reduction rules defined on λ-calculus expressions. Landin's extension of the λ-calculus necessitated the specification of an additional evaluation procedure since the reduction rules were no longer sufficient. In contrast to this approach, Strachey proposed to define programming language semantics in terms of pure λ-calculus expressions, thus eliminating the need for any additional evaluation mechanisms. Strachey handled assignments and jumps strictly within the confines of pure λ-calculus notation. Jumps were accommodated, for the most part, by textual rearrangement and use of the λ-calculus conditional. Assignments were modeled via the postulation of the primitive notion of a store, where a store represents a mapping from L-values (generalized addresses) to R-values (the contents of an L-value). Programming language constructs were then defined in terms of λ-expressions which mapped one store into another.

Another mathematical approach used in defining programming language semantics is based on Markov Algorithms. Markov Algorithms define a computation on a given input string in terms of a set of transformation rules which consist of left- and right-hand parts. If the input string contains a subsequence of symbols which corresponds to the left-hand side of a transformation rule, then the symbols on the right-hand side of this rule replace the subsequence of symbols in the input string. This process is repeated with the modified input string until no more replacements are applicable. One of the earliest proponents of this approach was van Wijngaarden (71), who proposed the use of extended Markov Algorithms for defining language semantics. His extensions included the use of metalinguistic variables in the transformation rules together with the notion of a dynamically growing set of rules. An application of this approach can be found in de Bakker's formal definition of ALGOL 60 (2). Related work in this area can be found in Caracciolo (8), Caracciolo and Wolkenstein (9), van Wijngaarden, et al. (72), and Ledgard (42, 43).

The formalism of algebraic theories has provided a rich source for semantic modeling. In a recent paper, Goguen and Thatcher (25) have shown that many apparently diverse approaches to semantics can be discussed within the framework of "initial algebra" semantics. They define an initial algebra as follows: Given a class of algebras $\underline{C}$, $S \in \underline{C}$ is initial if for each $A \in \underline{C}$ there exists a unique homomorphism $h_A: S \rightarrow A$. Within this definition, a formalized syntactic definition, S, can be considered an initial algebra for the class $\underline{C}$ of semantic algebras. A

semantic definition is provided with the specification of the unique homomorphism $h_A: S \rightarrow A$, where A is in $\underline{\underline{C}}$.

Knuth has outlined a pseudo algebraic approach for defining the semantics of context-free languages (39). His approach to semantic specification is tied directly to the syntactic representation provided by context-free grammars. He associates meaning or semantics with the terminals and nonterminals of the grammar. The semantics consist of both synthesized and inherited attributes, where synthesized attributes depend only on the descendents of nonterminals in the derivation tree and inherited attributes depend on the ancestors which appear in the derivation tree. The semantics of a string in the language is determined by the composition of the semantics of the terminals and nonterminals used in the derivation of the string. Subsequent work using Knuth's approach can be found in Wilner (78) and Fang (19).

A second algebraic approach, based on a lattice-theoretic formalism, has been proposed by Scott and Strachey (64, 65, 68). To a large extent, this approach is closely related to the Strachey λ-calculus approach discussed earlier. One of the objections to the λ-calculus semantic models was that there existed no set theoretic models for the λ -calculus. The difficulty in finding a set-theoretic basis for the λ-calculus is related to the paradoxical notion of the set of all sets. Scott demonstrated how a set-theoretic model for the λ-calculus could be constructed by restricting ones attention to only continuous functions (65). In a semantic model proposed by Scott and Strachey, computable functions are modeled by continuous functions defined on complete lattices. A system

state (S) is modeled as a complete lattice and language commands are modeled as producing state transformations [S → S]. These transformations are defined to be continuous functions on complete lattices. This class of functions is guaranteed to have minimal fixed points (Tarski, 69). The existence of these minimal fixed points guarantees the unique definition of semantics in this model. The Scott and Strachey approach has been used in specifying the semantics of QUEST (70), ALGOL 60 (53), and PL/I (4).

Quite recently, there has been a great deal of work undertaken in attempting to relate the notions of language semantics to the concepts of category theory. The intent here is to represent programming language models and the relationship between them as functors between various categories. The interested reader should consult Goguen, et al. (26) and Goguen and Thatcher (25) for extensive bibliographies of research in this area.

Axiomatic Models

In the axiomatic approach to semantic definition, memory states are no longer explicitly modeled. In this approach, emphasis is placed on what properties are satisfied by a memory state at a particular point in the execution of a program. The formalism of predicate calculus is used in stating these properties. In general, the semantics of a programming language command Q is expressed by an assertion of the form

$$P\{Q\}R$$

where P and R are propositional formulae representing properties of

memory states. The interpretation of this statement asserts that if P is true prior to execution of Q, and if Q terminates, then R is true after the execution of Q. Initial work in applying axiomatic semantics is attributable to Floyd (21). More recently, research in this area has been undertaken by Hoare (29), Hoare and Wirth (31), Cadiou and Manna (7), Manna (51), and Manna et al. (52).

Axiomatic semantics is particularly well suited to accommodate proof of correctness issues. The user can discuss his program in terms of assertions on memory states and hopefully can derive a proof of the program's correctness in terms of assertions on the final memory state.

Operational Models

There are two basic approaches for specifying the operational semantics of a programming language: interpreter-based models and compiler-based models. Both of these approaches are normally syntax driven in the sense that actions are dictated by the recognition of syntactic entities. In an interpretive approach, the recognition of a syntactic unit in the source program is directly associated with a corresponding memory state transformation. In a compiler-based model, the recognition of a syntactic unit in a source program written in language L is associated with a language construct from some target language L'. The "execution" of the L' construct, subsequent to the complete translation of the L source program, produces the desired memory state transformation. In general, there are two methods of defining how the execution of the L' program will proceed. This execution can be specified by an interpreter-based model for L' or L' can be defined to consist of primitive

constructs which require no further definition. In either event, compiler-based approaches are justifiably criticized on the grounds that they provide only a restatement of the semantic definition problem, rather than a solution (77). In this survey the compiler-based models of Feldman and Wirth and Weber and the interpretive models proposed by McCarthy, the IBM Vienna group and Johnston will be discussed.

As previously mentioned, the unique characteristic of the operational approach is the use of information structures to explicitly model memory states. In this sense, operational models are representation-dependent definitional frameworks. The type of information structure used in an operational model provides one criterion for classifying an operational scheme.

McCarthy (47, 48), the founder of operational semantics, used the notion of a state vector as the information structure in his approach. A state vector is defined to be a list of ordered pairs of the form (identifier, value). At any given point t during execution, the state vector contains the identifiers known at time t together with the values currently associated with those identifiers. McCarthy uses two primitive functions to access and update state vectors. The semantics of a language construct is expressed via the state vector transformation that it induces. McCarthy has used the state vector approach to define the semantics of a subset of ALGOL 60 (49). This subset did not include block structure, thus insuring that the identifiers in the state vectors were unique.

The development of the programming language PL/I was accompanied

by the development of a definition language for the language. This definition language, the Vienna Definition Language (VDL), uses labeled trees as its basic information structure. In the VDL model, an interpreter state is usually modeled as a tree with three labeled components: a text component, a memory component and a control component. The first step in deriving a VDL semantic specification is to convert programs written in the source language (concrete programs) into abstract representations or "abstract syntax". Programs represented in "abstract syntax" form have a tree structure in which the labels on the edges are used to access particular instructions. The abstract syntax representation of the program is held in the text component of the interpreter state. The memory component of the interpreter state is used to hold tree representations of the data values used in a computation. The control component contains information which specifies how instructions in the abstract program map one interpreter state to another. The semantics of a language construct is defined in terms of its effect on the interpreter state. The defining document on the VDL approach is Lucas, et al. (45). Other papers demonstrating the applicability of VDL include Henhapl and Jones (27), Lucas (44), Lucas and Walk (46), Walk, et al. (73) and Wegner (77).

The Contour Model proposed by Johnston (35), has a two dimensional tree-like information structure base. It has been claimed that this model is at least as powerful as the VDL approach and is more amenable to a discussion of implementational issues (5). The Contour Model is based on the concepts of nested block structure, accessing environments, labels and cell retention. In this model, computations are represented

by sequences of snapshots, where each snapshot is a data structure (contour configuration) consisting of cells and their contents. The Contour Model is most effective in modeling computations expressed in block structured languages. A contour is allocated upon block or procedure entry and deallocated (unless retention is indicated) upon exit. The nesting of the contours reflects the block structured nature of the computation. The relation between source programs and their semantics is specified by defining an interpreter which transforms source language commands into operations on contours. The use of the Contour Model in defining the semantics of the programming language Oregano is presented in Berry (5). Recent theoretical development of the model is presented in Johnston (36, 37) and Johnston et al. (38).

The notion of defining the semantics of a language in terms of a standard compiler for the language was originally proposed by Garwick (22). In attempting to realize this objective, Feldman (20) proposed a Formal Semantic Language (FSL) which defines a high-level programming language in terms of code generation routines defined on an abstract computer. The FSL primitives, e.g., JUMP, MULTIPLY, ASSIGN and PLUS, are not defined. As a result FSL can be criticized on the basis that the defining language is more complex than necessary and very much in need of semantic definition itself. This objection was satisfied in the Wirth and Weber approach to compiler-based semantic definition (79, 80). Wirth and Weber defined the semantics of Euler in terms of elementary operations on a pushdown stack machine.

Semantics of Data Structures

The semantic models discussed up to this point have been concerned with the definition of the semantics of programming languages. A great deal of work has also been undertaken in developing models which define the semantics of data structures. Since it is the author's contention that there is no clear cut separation between research in programming languages and research in data structures, it is appropriate to include a discussion of research in data structure semantics in this survey.

Rosenberg (59, 60) has proposed a formalism called the Data Graph Model in which the semantics of data structures can be expressed. A data graph is obtained from a data structure by ignoring the specific data items which appear at the nodes of the structure and concentrating only on the linkages in the structure. Rosenberg has attempted to relate the logical concept of a data structure to its physical representation, i.e., its implementation in a computer. In particular, he uses the data graph representation of structures to detect structural uniformities which can be exploited in implementing the accessing of nodes within a computer. Rosenberg has characterized two classes of data graphs which are realizable by two distinct implementation mechanisms. Rooted or addressable data graphs are those which are implementable via relative addressing while free-rooted or uniformly translatable data graphs admit of relocatable implementations. An application of this model in analyzing the properties of data graphs which correspond to extendible arrays can be found in Rosenberg (61-63).

Earley (14, 15) has suggested that there are three hierarchical levels at which data structures should be discussed: the relational level, the access path level and the machine level. The relational level represents the most abstract level in this hierarchy. At this level a mathematical description of the structure is given along with a specification of how the structure may be manipulated. This level does not describe the access paths that will be used nor how the structure may change. The access path level makes explicit the access relationships that exist between the individual data items, but it does not specify how these access paths should be implemented. Earley uses a V-Graph Model to describe structures at the access path level. In a V-graph, the nodes represent parts of the structure and the links represent access paths. Finally, at the machine level, the actual implementation of a structure on a machine is discussed. This multilevel approach to data structure analysis is consistent with the stepwise refinement philosophy of structured programming. Earley has implemented some of his ideas in the VERS language (16).

A purely operational approach to specifying the semantics of data structures has been proposed by Ellis (17). Ellis' work is based on the common base language interpreter proposed by Dennis (12). The common base language model is an interpreter-based operational semantic model for programming languages. Ellis uses this framework to discuss the ways in which programming languages deal with data structures.

Semantics of Name Accessing

A very small number of semantic models have emphasized name accessing descriptions. Two such models are considered in this survey: namely, those proposed by Gilmore (24) and Boyle and Grau (6).

The Gilmore model is based on an abstract computer with three primitive operations: an initial load operation, a lambda removal or function application operation and a conditional operation. The initial load operation has the form "a:b", where "a" and "b" are strings. The effect of executing such an operation is the placement of the string "b" into a memory location corresponding to the string "a". The initial load operation thus has the effect of establishing naming relationships between a location string name and the contents of the location. A subsequent reference to the string "a" will denote the value contained in location a.

The lambda removal operation is quite standard. If the string "g" denotes a location containing the string "(lambda, y, (lambda, x, y(y(x))))", then "g(f)" will result in an application of the lambda removal operation, i.e., the bound variable of the contents of g will be replaced by "f" giving "(lambda, x, f(f(x)))".

The conditional operation has four arguments and is defined as follows:

$$\text{cond}(a1, a2, a3, a4) = \begin{cases} a3 \text{ if } a1 = a2 \\ a4 \text{ otherwise} \end{cases}$$

Higman (28) has extended some of Gilmore's original ideas and defined a Gilmore system to be one in which the following hold:

1) A name is a string of characters. A name may be either valid, invalid or intrinsic.

2) A name may possess (potentially) an intermediate value and a final value.

3) A result is defined to be the final value of a program, where a program is a name (character string).

4) A string may denote a literal and there are rules for deciding when this is the case. Literals denote themselves and therefore have immediate final values.

5) A definition is a subsequence of a program. The semantic effect of a definition is that a certain literal is the final value of a certain name.

6) If a name cannot be assigned a final value under (4) or (5), there are rules for transforming a string into its intermediate value. The result of this transformation is a sequence of names consisting of one operator and one or more operands.

7) There are rules for evaluating an (operator, operand(s)) grouping and obtaining a result.

8) If the final value of a name is not immediate, the final value is recursively defined to be the result obtained by applying the operator to the final values of its operands.

9) A name is valid if a final value can be derived from it. An intrinsic name has no final value but it is permitted for its use as an operator. All other names are invalid. The set of all valid names under a given set of rules constitutes a language.

To the author's knowledge no additional work has been done with Gilmore's system. His original model accommodated only functional languages, but Higman (28) did include an extension to handle imperative assignment. A more serious objection to the Gilmore system is that naming clashes were avoided by simply prohibiting their occurrence in legal programs. This is a somewhat restrictive assumption that obscures the complexity of the name accessing problem. The accommodation of naming clashes is handled directly in the Boyle and Grau model.

The Boyle and Grau algorithmic semantic model for ALGOL 60 identifiers (6) provides, via a set of algorithms, a mapping from pure ALGOL 60 programs to $ALGOL_1$ programs. In $ALGOL_1$ no identifier is ever redeclared and therefore each identifier is unambiguously associated with its proper declaration.

A mapping, Φ, from ALGOL 60 to $ALGOL_1$ is defined as a composite mapping consisting of $\Phi_1 \circ \Phi_2 \circ \Phi_3 \circ \Phi_4$ where

Φ_4: ALGOL 60 $\rightarrow$ $ALGOL_4$,

Φ_3: $ALGOL_4$ $\rightarrow$ $ALGOL_3$,

Φ_2: $ALGOL_3$ $\rightarrow$ $ALGOL_2$, and

Φ_1: $ALGOL_2$ $\rightarrow$ $ALGOL_1$.

Mappings φ_1, φ_2, φ_3 and φ_4 are defined respectively to be the projections of Φ_1, Φ_2, Φ_3 and Φ_4 which handle the mapping of identifiers. φ_1 and φ_2 are concerned with the static (compile-time) resolution of identifiers while φ_3 and φ_4 involve the dynamic (run-time) resolution of names. A pure copy rule semantic form is used in the resolution of the locals of

procedures for recursive calls.

The Boyle and Grau model is quite exact and very detailed in its handling of indentifier resolution. Perhaps the main drawback of this approach is that it is too detailed and therefore complicates rather than illustrates the semantics of name accessing.

Applicability of Formal Semantic Models

At the beginning of this chapter several objectives for formal semantics approaches were listed. The remainder of the chapter is devoted to a brief evaluation of the various semantic approaches that have been surveyed in terms of whether or not they meet those objectives.

It should be apparent that, to date, no "super" semantic model has been developed. The term "super" is used here in the sense that it satisfies all objectives for designers, implementors, users and theoretical computer scientists. It may very well be that "super" is a myth, an unobtainable goal, and our efforts would be more fruitfully rewarded if we attempted to develop models for individual rather than conglomerate groups. In any event the following critique should be made before proceeding.

The mathematical models have been quite popular with the computer scientist. This is true despite the fact that, in general, the mathematical models have not met the primary objective of the computer scientist, i.e., applicability to a large class of programming languages. The popularity of this approach with the computer scientist is probably due to the fact that the rigor of the mathematical approach has been

used to support the legitimacy of theoretical computer science research. The author is of the opinion that the legitimacy of this type of research is independent of mathematical parentage and advances the proposal that mathematical rigor should be employed only as a tool and not for its name dropping effect.

Since mathematical models tend to abstract out implementation details, these models tend to support the objectives of the language designer. This approach seems well-suited to the detection of ambiguous, incompatible or contradictory language constructs.

In emphasizing implementation-dependent models, the operational approach has been quite appealing to the language implementor. The use of information structures is quite appropriate for the implementor. Although this approach has not been used widely by theoretical computer scientists, it seems that operational models do offer a tremendous advantage in that they highlight rather than abstract out the objects that the computer scientist wants to study.

No approach has yet offered the user a concise, readable and transparent description of a complicated programming language. The user has been able to derive some benefit from the axiomatic models in formalizing and proving properties about programs, but even this benefit is useful to only a small number of users.

The data structure and naming models discussed in this chapter have not been adequately tested as to their applicability to a wide class of problems. Both Rosenberg and Earley have achieved limited success in dealing with implementational questions however.

In Chapter III, an operational approach to name accessing modeling is presented. This model attempts to satisfy some of the objectives of language implementors and computer science researchers. The relative merits of the proposed model are discussed in Chapter VI.

CHAPTER III. THE ACCESSING GRAPH MODEL

Basic Name Accessing Concepts

Consistent with some of the more recent definitions of names in programming languages (18, 55), a rather general interpretation of a name will be used in this work. Specifically, a name is defined to be a syntactic construct used to symbolize or represent an object. Within this definition the following are legal names.

X	simple name
X + Y	expression
13	literal
A[3]	subscripted name
+	simple name

The simple name "+" corresponds to an intrinsic name as defined in a Gilmore system. Intrinsic names normally have a fixed interpretation as defined in the language and are therefore relatively simple to handle. The same is true of literals, which denote themselves. Literal names and intrinsic names will be considered no further in this discussion.

Names such as "X + Y" and "A[3]" are composite names and defined in terms of their components. With this in mind, this discussion will revolve around their primary components - simple names. With the exclusion of intrinsic and literal names, simple names usually include the following:

1) names for simple variables,

2) names for data structures,

3) names for subprograms,

4) statement labels, and

5) formal parameters.

Using a generalized definition of programming language names, Pratt (55) has isolated five operations normally available in a programming language for manipulating names. These operations are naming, unnaming, activating, deactivating and referencing. Naming is defined to be the act of creating an association between an identifier and a program or data object. Unnaming denotes the act of destroying such an association. Activating is the operation of making an existing association known or active and therefore available for use in referencing which is the act of retrieving the data object or program currently associated with an identifier. Deactivating similarly refers to the act of making inactive a particular association.

The Pratt primitives are fairly typical of the current view of operations available for manipulating names. A more generalized set of primitives is adopted in this work.

Primitive Notions About Naming

This dissertation is primarily concerned with the semantic definition of the naming conventions of programming languages. Accordingly, attention will be focused on those aspects of program execution related to the creation and manipulation of names. In particular, in specifying the semantics of a given programming language, emphasis will be placed

on the naming mechanisms of the language, those aspects of the interpreting computer that accommodate the naming mechanisms, and the components of the underlying data structure that are used in the support of the naming mechanisms. These notions will now be presented in some detail.

In this section, a metalanguage is presented in which name accessing issues can be discussed. This metalanguage is first expressed in terms of primitive functions needed to accommodate the name accessing features explicated in this dissertation. Subsequent to this presentation, these primitive functions are used to develop a name accessing language, NAL, which will serve as a target language for the translation of high-level name accessing features. The semantics of NAL will then be described in terms of the Accessing Graph Model, AGM.

In one sense, NAL is the "machine language" for an abstract computer which interprets the naming mechanisms of the language being modeled. The semantics of a particular naming mechanism will be defined in terms of a NAL program (whose semantics are defined in terms of the AGM). The semantics of a given source language program will be given in terms of a NAL program whose execution then specifies the precise behavior of the names created and manipulated during execution of the source program.

The main objects manipulated by NAL programs are names. Throughout this dissertation a name (in the abstract) will be viewed as a finite set of ordered pairs called property-value pairs. It is assumed that a name has some (finite) number of properties that are of interest and that each property assumes a value from some set of values. If a name, n, is represented as the set of ordered pairs $\{(P_1,V_1), (P_2,V_2), \ldots, (P_m,V_m)\}$,

then we say that n possesses property P_i with value V_i for i = 1, 2, ..., m.

For the purposes of this discussion, it is assumed that properties may possess one of two types of values: simple values or subname values. A subname value is a name that is the value of a property of a name.

Using the (P,V) descriptors, a name in a program can be described by those properties that are of interest. For example, a given name might be represented as {('symbol', X), ('type', integer), ('value', 3)}. This name then is being viewed as having three properties of interest: a 'symbol' property whose value is the string "X", a 'type' property whose value is the type integer, and a 'value' property whose value is the number 3.

In defining a naming system, it is necessary to establish rules for avoiding naming conflicts. If we assume that any naming system creates a universe N, called the set of names, then the usefulness of the naming system stems from its ability to select elements from this universe. This selection process is more commonly known as name accessing and is accomplished via an interrogation of the (P,V) pairs associated with the elements of N. While it is possible for more than one name to possess the same (P,V) pair, the integrity of a naming system stems from the fact that a given collection of (P,V) pairs can be associated with only one name at a time. This uniqueness requirement insures nonambiguous name accessing.

It will be assumed that when a name is created there will be certain (P,V) pairs given to that name and that these (P,V) pairs will remain

unchanged throughout the lifetime of the name (i.e., until the name is destroyed). Such properties will be called intrinsic properties of the name. All other properties associated with the name during its lifetime will be referred to as secondary properties. Note that it is a name's intrinsic properties that guarantee its uniqueness and it is the intrinsic properties of a name that are interrogated in selecting an element of N.

Example 1:

Assume that S is a naming system in which the following properties can be associated with names:

1) A 'symbol' property whose value is a string of alphabetic characters,
2) A 'type' property whose value is either an integer type or a real type, and
3) A 'value' property whose value is either an integer number or a real number.

Now, suppose that in S it is stipulated that when a name is created it is given a particular 'symbol' value and a particular 'type' value with the constraint that these values cannot be changed during the lifetime of the name. Then the 'symbol' property and the 'type property are both intrinsic properties of names in S and the 'value' property is a secondary property. Note that in S two different names may have the same 'symbol' property but then must have different 'type' properties.

It is postulated that a naming system possesses the following basic capabilities:

1) The ability to create names,

2) The ability to attach and update secondary properties of names,

3) The ability to retrieve the value of any property of a name,

4) The ability to remove secondary properties of names,

5) The ability to access names on the basis of their properties, and

6) The ability to destroy names.

These capabilities are considered in detail below.

Name creation

The creation of a name implies the establishment of the intrinsic properties of that name. These intrinsic properties will be treated as "sacred" in the sense that it will be impossible to remove these properties without destroying the name. The creation of a name will be accommodated with a createname primitive. This primitive will take one argument, a list of (P,V) pairs. The effect of this primitive is to add to the set N (the universe of names) a new name with the specified properties as intrinsic properties. In the event N already contains a name whose list of (P,V) pairs is identical with that specified for the newly created name then these two names are indistinguishable.

Attaching and updating properties

The attaching and subsequent modification of secondary properties are handled by an attachprop primitive. This primitive takes two arguments: a name n and a list of (P,V) pairs. The effect of the attachprop primitive is to add the specified secondary properties to the name n or to update any of the specified secondary properties that already exist

for the name. Any attempt to update an intrinsic property will have a null effect.

The effect of the attachprop primitive is dependent on the type of value that is being attached. For attachprop (n,{(P,V)}), there are two possibilities: V is a simple value or V is a subname value. If V is a simple value, a copy of this value is attached as the P value of n. If V is a subname value, V itself becomes the P property of n.

Property removal

Property removal is handled by a removeprop primitive. This primitive takes two arguments, a name and a list of properties, and its effect is to remove the specified properties from the given name. Any attempt to remove an intrinsic property or a nonexistent property will have a null effect.

Name accessing

Elements of N can be grouped on the basis of common properties. The accessing of a name is accomplished on this basis. Given a set of names S, a composed property P, and a property value V then propset (S, P, V) returns as its value the set consisting of those names in S for which the composed property P has value V. More precisely, if P_1, P_2, ..., P_k are properties then $P_1.P_2.\cdots.P_k$ is a composed property and we define propset as propset (S, $P_1.P_2.\cdots.P_k$, V) =
$\{y \mid y \in S \wedge$ propval (...propval (propval (y,P_1), P_2)...P_k) = V$\}$. An "*" can be used for V in the event the property value is not of interest. Note that k>1 implies that the value of P_j, j< k must be a name.

Also, given a set of names S, a composed property P, and a value V then propx (S, P, V) returns as its value that one element of S whose composed property P has value V. If no elements of S or more than one element of S satisfy this criterion then an error will result. As with propset an "*" can be used for V.

Value retrieval

The retrieval of the value of a specified property of a name is accomplished with a propval primitive. This primitive takes two arguments, a name and a property, and returns the corresponding value. Any attempt to retrieve the value of a nonexistent property will result in an error.

Name destruction

The destruction of names is accomplished with a destroy primitive. This primitive takes one argument, a set of names, and has as its effect the removal of those names from the universe N.

The seven primitives: createname, attachprop, propval, removeprop, propset, propx and destroy constitute the basic instruction set for the name accessing language NAL. In addition to these primitives NAL supports three special functions which operate on simple values which happen to be integers. These functions are '+', Max and Min.

The '+' function simply allows for the summing of two integer values. If a set of names S is defined such that each element of S has a certain property P which possesses an integer value then

propset (S, P, Max) returns those elements of S with the largest P value, and

propset (S, P, Min) returns those elements of S with the smallest P value.

Having defined the NAL instruction set, the Accessing Graph Model is now introduced to serve as a descriptive device for illustrating the execution of NAL programs.

Description of the Accessing Graph Model (AGM)

The motivation for the AGM stems from the author's belief that name accessing relationships can perspicuously be modeled in a graph-theoretic framework. A similar assumption has previously been made with respect to the modeling of data structures as can be noted in the works of Earley (14-16) and Rosenberg (59-63). For example, Earley (14, p. 618) has stated that "Our fundamental notion about the semantics of data structures is that it may be represented by directed graphs with names on the edges." Rosenberg (59, p. 193) has elaborated even further on this point when he stated that "A data structure can be viewed as a collection of primitive data items in conjunction with a set of relations on these items. Such structures can fruitfully be represented as directed graphs with nodes labeled by data items and with each edge from node n to node m labeled by the relation which holds between the data item in node n and that in node m."

Both Rosenberg and Earley have applied graph-theoretic modeling techniques to data structures with some success. Their objectives were twofold. First, they were interested in providing a descriptive model

which would be useful in specifying the semantics of data structures. Secondly, they wished to use the graph-theoretic descriptions to gain insight into potential implementation algorithms. In applying graph-theoretic modeling techniques to name accessing issues, it is hoped that similar success might be achieved in both describing and implementing the name accessing features of programming languages.

In the AGM, a program, together with the data it operates on, are viewed as a composite data structure and modeled as a directed graph called an accessing graph. Program components are modeled as program nodes (represented by boxes) and data components are modeled as data nodes (represented by circles). Relations which hold between the nodes of an accessing graph are modeled as labeled edges between the nodes. These relationships belong to one of the following three classes.

1) program node $\approx$ program node

Example 2

In a program containing the following lines of code:

$a \leftarrow a + b;$

$b \leftarrow a;$

certain relationships exist between these lines of code. In particular, the second statement is the successor of the first and the first is the predecessor of the second. In the AGM these relationships are modeled by a predecessor edge (π) and a successor edge (σ) as shown in Figure 2.

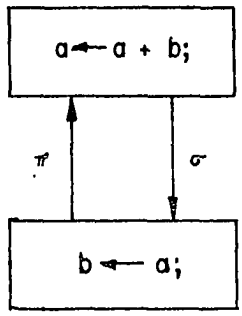


Figure 2. Program node ≈ program node relationships

A few remarks are in order about the nature of the π and σ edges. It is intuitively appealing, and not completely incorrect, to consider the σ edge as a flow of control relationship. In other words, given program nodes A and B together with a σ edge from A to B, we could assume that after the "execution" of A, node B would be "executed". This assumption would be valid of course only if the "execution" of A did not result in a transfer to some other program node. In any event, it is quite easy to argue the validity of σ type edges on the grounds of normal flow of control. The π edge doesn't conjure up a nice intuitive meaning however. In part, this is due to the fact that program representations (e.g., flow charts) which appear in the literature normally model only σ type edges. The significance of the π edges is argued on a semantic basis.

Program components and even entire programs do not normally have an independent meaning. Consider the statement

$$A \leftarrow A + B; \ .$$

Before any attempt is made to define the meaning of this statement, the history or context of the statement must be considered. Have A and B been declared? Do A and B have well-defined values? These questions can be answered only if we know which statements preceded this statement in the program. Even the program itself may not answer all of the questions. For example, the meaning of '← ', '+' and ';' are not defined in the program, but in a context which precedes even the program. In summary, it is assumed that program nodes have both a future (σ) and a past (π). The interested reader should consult Knuth (39) for a similar argument about the necessity of both inherited and synthesized attributes of context-free grammars.

2) program node ≈ data node

Our fundamental assumption concerning the interrelations of program nodes and data nodes is that program nodes create and reference data nodes and data nodes return requested data values to program nodes. Given an accessing graph configuration, the relationships existing between the program nodes and the data nodes constitute the name accessing relationships of the system. Normally, these relationships are acquired or brought into existence dynamically so it is necessary to specify a "point of perspective" with respect to an accessing graph. The point of perspective tells us which program nodes have been visited and therefore the naming relationships which have been acquired. In future discussions, the point of perspective will be specified in terms of a path from some designated root node.

Example 3

Consider the accessing graph in Figure 3. The shaded node is used to denote the root of the graph and the point of perspective, denoted by P, is specified as P = σσσσσσ or σ^6 with respect to the root node. In other words, the accessing graph in Figure 3 is being viewed from the bottom node. The significance of the point of perspective and the visitation of nodes will be discussed in greater detail later in this chapter. The actual contents of the program nodes have been left unspecified in Figure 3. It is however assumed, that in obtaining the point of perspective P, the naming relationships for the names X, Y, Z and W have been acquired.

In the AGM, simple values are represented by placing the value representation within a circle. For example, the value of the 'SYMBOL' property of the first name encountered in Figure 3 is the string "X". Names or subnames are actually a set of (P,V) pairs. In the AGM, these objects are modeled by header nodes, denoted (H), with a finite set of outgoing edges representing the properties of the name or subname. The object pointed to by the outgoing edge P_i is the value of the P_i property. Those program nodes that have outgoing edges to header nodes are said to possess a name relationship with those header nodes. Subsequent references to an instance of a name will be made through that program node which possesses a name relationship with that instance of the name. The ρ edges from the header nodes to the program nodes indicate the returning capability of data nodes, i.e., the fact that they return values to referencing program nodes.

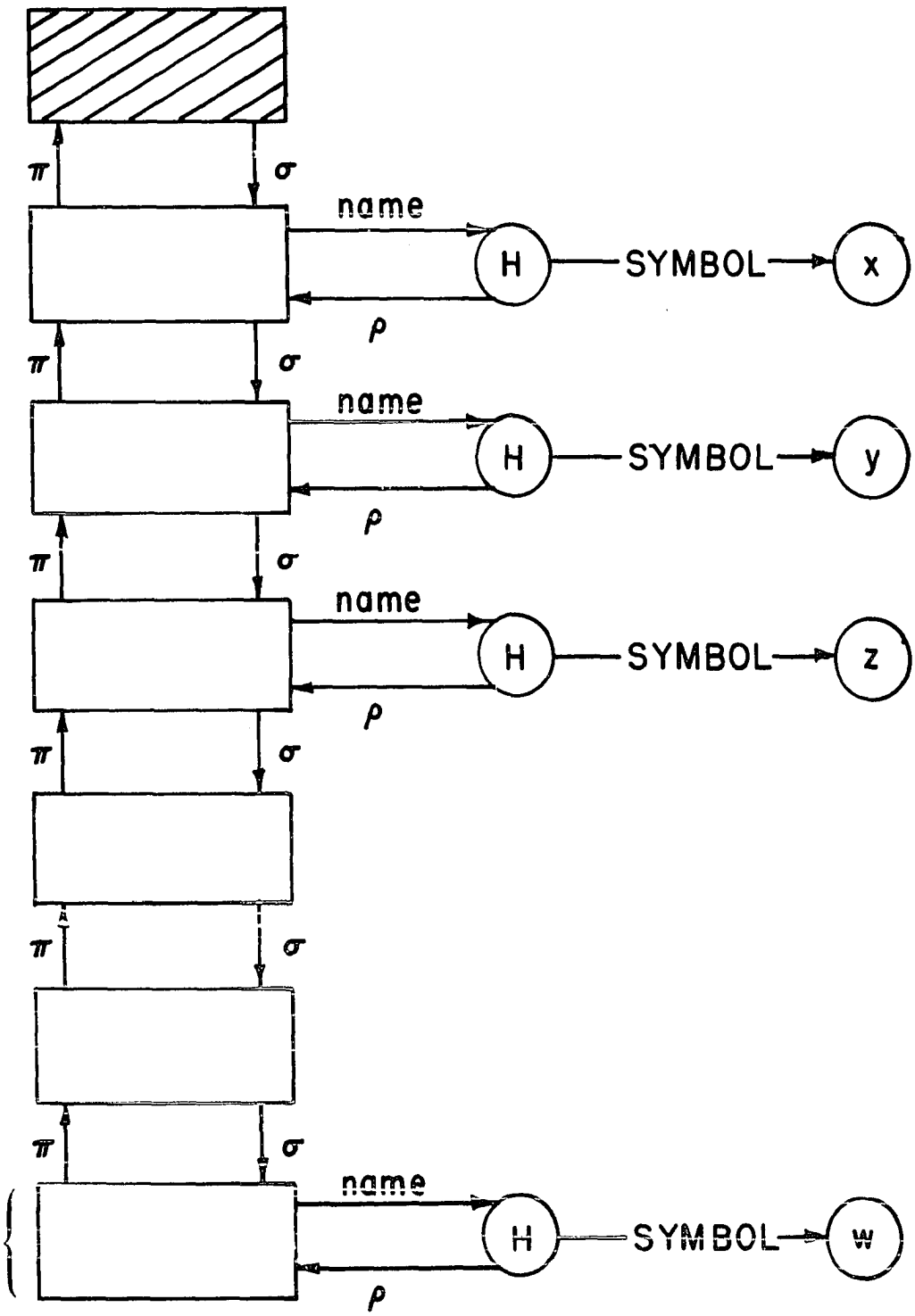


Figure 3. Program node ≈ data node relationships

3) data nodes ≈ data nodes

Example 4

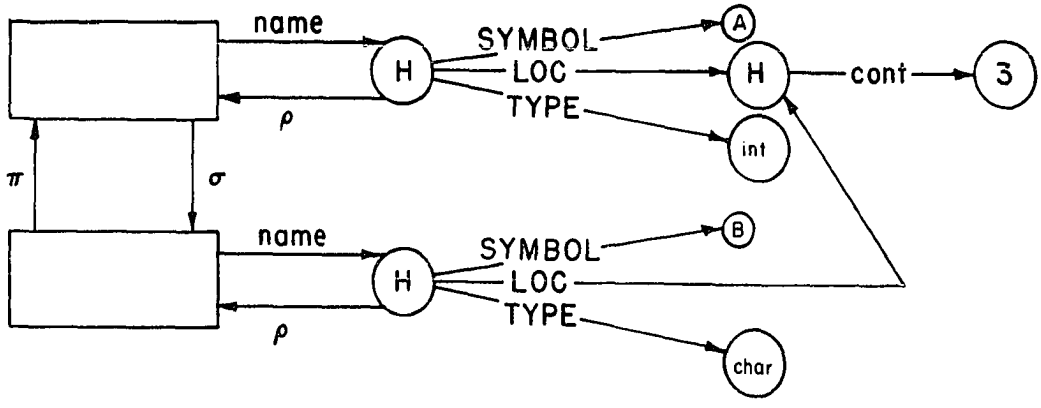


Figure 4. Data node ≈ data node relationships

In the accessing subgraph depicted in Figure 4, a sharing relationship exists for the 'loc' properties of the two names illustrated. One interesting aspect of this example is that the 'type' property is associated with the name rather than the subname. As a result the 'cont' property value, "3", can be interpreted either as a integer or a character string depending on the access path that is used. This type of sharing can be effected with the PL/I DEFINED attribute.

The data node to data node relationships have been closely examined in the literature. These relationships were an object of study in both Rosenberg's and Earley's work. The relationships between program nodes and program nodes (flow of control issues) have also been extensively

examined. The relationships that exist between program nodes and data nodes are the main concern of this dissertation. These relationships constitute the crux of the name accessing problem in that they relate the names used in the program text to the denotations which the names refer to, i.e., the objects of the program operates on.

Accessing graphs model program-data composites. These graphs are capable of being "executed" by an abstract computer, the AGMC. AGMC is effectively a function which maps one AGM state to another, where an AGM state is an ordered pair (g, p); g representing an accessing graph and p a point of perspective in g. In mapping a state (g, p) to (g', p') AGMC may modify g by manipulating either the program nodes or data nodes of g. The transformation of p to p' reflects the flow of control process.

In executing programs, the AGMC makes use of the underlying accessing graph structures. During program execution, this graph structure will contain a representation of the submitted program (and data) as well as any other information required to support the language. Execution proceeds in a step by step fashion with each step being characterized by a transformation on this underlying graph structure. The effect of program execution is represented by the sequence of transformations made (by the AGMC) on the underlying graph structure.

Before proceeding with the use of the AGM in modeling name accessing, the general modeling technique of this approach should be explicated. This technique is described as follows:

1) Let p be a source language program (written in a language L) whose semantics we wish to specify.
2) A TRANSLATE function is defined for L such tnat TRANSLATE (p) produces the corresponding AGM representation of p.
3) An EXECUTE function is then defined to interpretively execute the AGM representation of p. The EXECUTE function is defined in terms of NAL code.
4) In applying the name accessing primitives, attachprop, propset, propx, etc., the universe N is defined to be those names that are reachable along the π chain from the point of perspective.

CHAPTER IV. APPLICATIONS OF THE MODELING TECHNIQUE

The purpose of this chapter is to analyze various name accessing capabilities that can be found in current high-level programming languages. A mini-language approach has been adopted for achieving this goal. Using this approach, four min-languages have been developed which highlight the following name accessing features: 1) accessing in a nonblock structured environment, 2) block structure with default scoping, 3) block structure without default scoping, and 4) procedure closures. For each mini-language a syntactic description of the language, and an informal discussion of its translation into AGM form is provided. Execution of an AGM representation in terms of NAL code is described for the general case. These notions are illustrated by the translation and execution of a representative program written in each mini-language

Mini-Language 1 (ML-1)

ML-1 is a very simplistic language. A ML-1 program consists of a sequence of declaration statements followed by a sequence of assignment statements. The end statement is used to flag the end of a program. In ML-1, it is required that a given identifier be declared at most once and that only declared identifiers may be used in the assignment statements. A syntactic description of ML-1 is presented in Figure 5.

A simple name in ML-1 may possess three properties: a 'SYMBOL' property whose value is its identifier representation, a 'val' property whose value is the current integer number associated with the name, and

a 'ρ' property whose value is the program node that created the name. The 'SYMBOL' property is an intrinsic property of a name in ML-1. Type has not been included as a property since all name, by default, possess the same type, i.e., integer.

P::=D;S <u>end</u>

D::=D;δ|δ

δ::=<u>declare</u> identifier

S::=S;α|α

α::=identifier $\leftarrow$ term

term::=identifier|integer

Figure 5. Syntactic description of mini-language 1.

Program nodes in ML-1 normally have two properties: a 'π' property whose value is the predecessor program node, and a 'σ' property whose value is the successor program node. In addition to these two properties, which are inherited via translation into AGM form, a program node may acquire, via execution, an additional property; a 'name' property whose value is the simple name created by that program node.

The translation of a ML-1 program into AGM form is very straightforward. Each declaration and assignment statement corresponds to a program node in the AGM program representation as does the ML-1 <u>end</u> statement. Each program node acquires, via translation, a 'π' property and a 'σ' property as described above.

The execution of the AGM representation proceeds as indicated below.

Declaration node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "declare identifier;".

NAL code:

attachprop (σ^j, 'name', createname ({('SYMBOL', identifier)}));

attachprop (propx (N, 'SYMBOL', identifier), {('ρ', σ^j)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "identifier" and a 'ρ' property whose value is σ^j.

Assignment node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "identifier1 ← identifier2;".

NAL code:

attachprop (propx (N, 'SYMBOL', identifier1), {('val',

propval (propx (N, 'SYMBOL', identifier2), 'val'))});

If the contents of σ^j is "identifier1 ← number;" the corresponding NAL code would be

attachprop (propx (N, 'SYMBOL', identifier1), {('val', number)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that the

name whose 'SYMBOL' property is "identifier1" has acquired a new or updated 'val' property whose value is the current value of the 'val' property of the name with 'SYMBOL' property "identifier2" (or the number in the case of the simplified assignment form).

End node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "end;".

NAL code:

destroy (N);

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that all program nodes lose their name properties.

Example 5

ML-1 Program

```
declare a;
declare b;
declare c;
a ← 1;
b ← 2;
a ← b
end
```

An execution trace of the ML-1 program listed above is given in Figures 6a-6h. In all traces a left set bracket "{" has been used to flag the next node that will be executed.

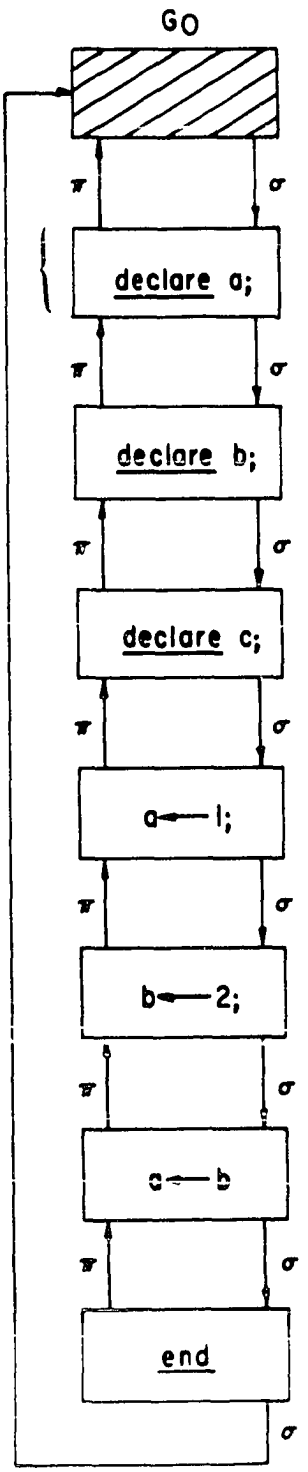
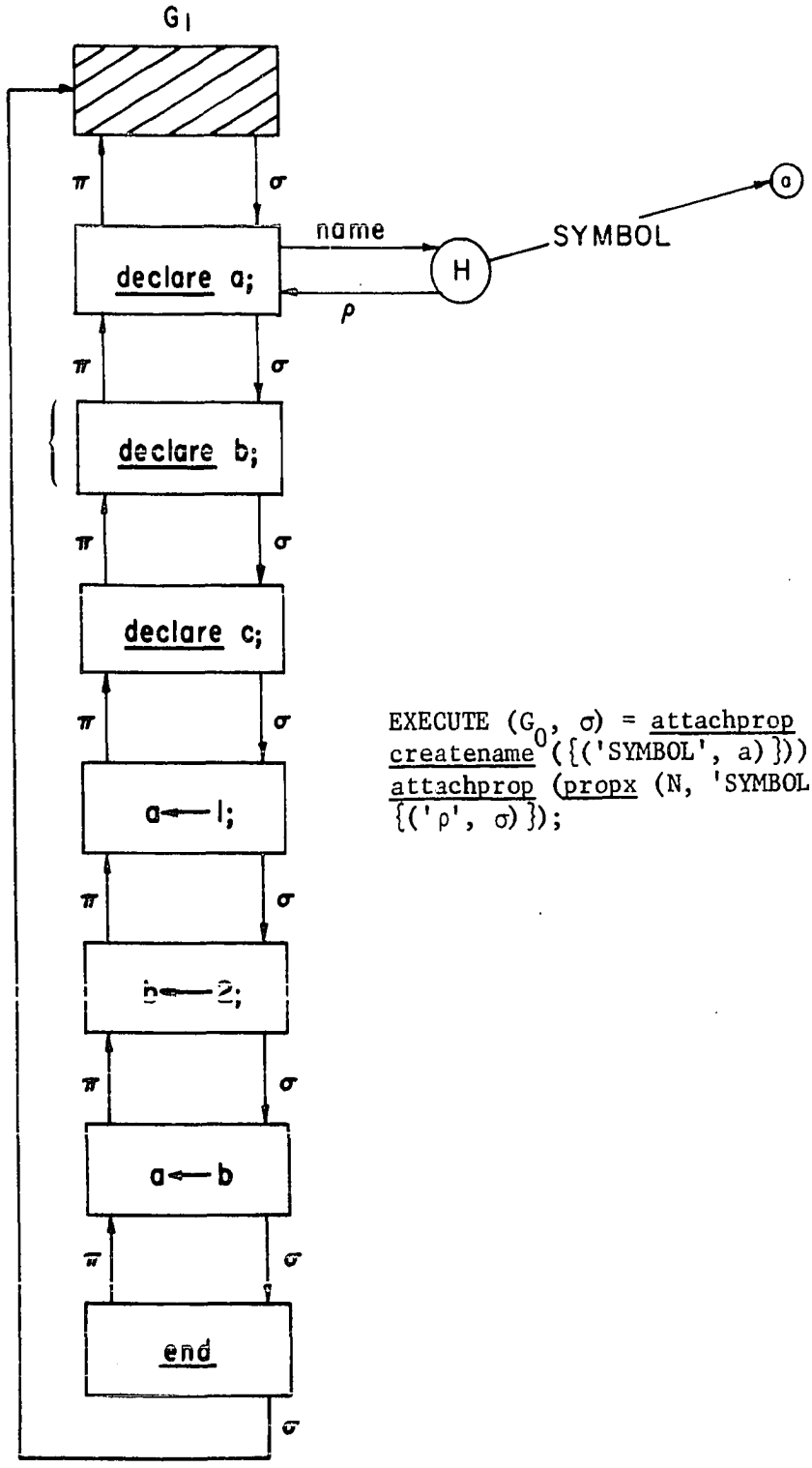
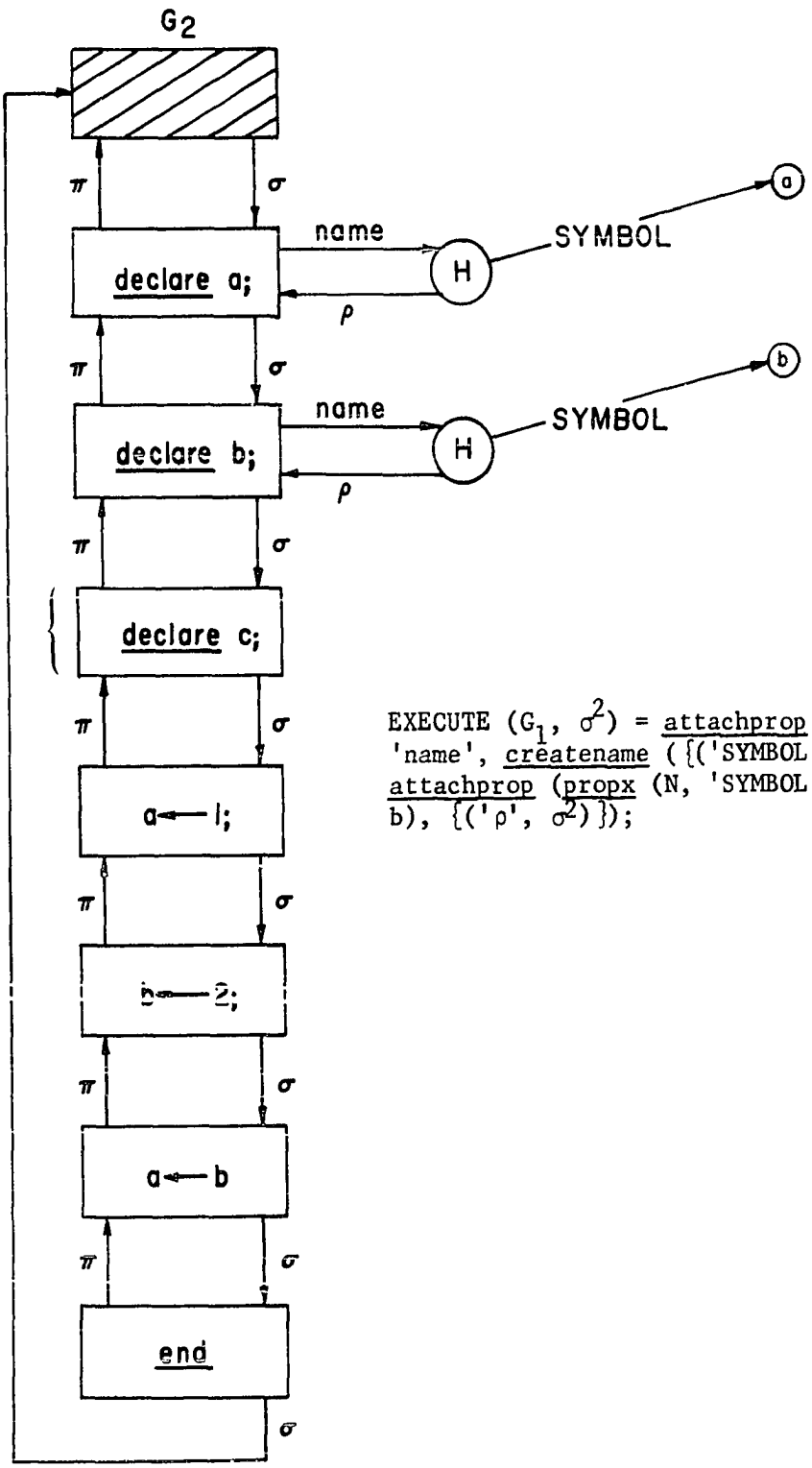


Figure 6a. Initial AGM representation for example 5 with state = (G_0, σ)



EXECUTE (G_0, σ) = attachprop (σ, 'name', createname ({('SYMBOL', a)}));
attachprop (propx (N, 'SYMBOL', a), {('ρ', σ)});

Figure 6b. Execution trace for example 5 with state = (G_1, σ^2)



EXECUTE (G_1, σ^2) = attachprop (σ^2, 'name', createname ({('SYMBOL', b)}));
attachprop (propx (N, 'SYMBOL', b), {('ρ', σ^2)});

Figure 6c. Execution trace for example 5 with state = (G_2, σ^3)

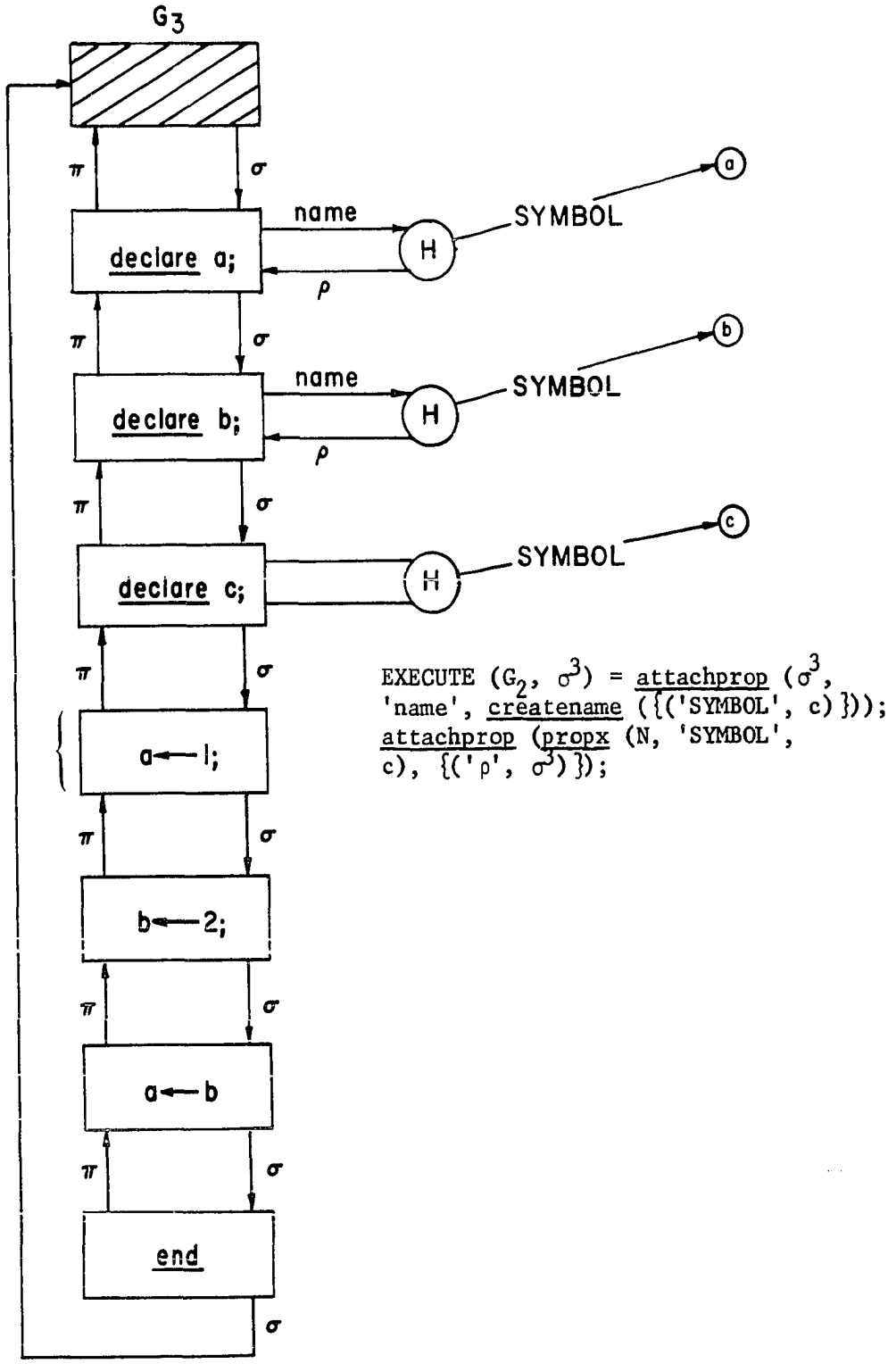


Figure 6d. Execution trace for example 5 with state = (G_3, σ^4)

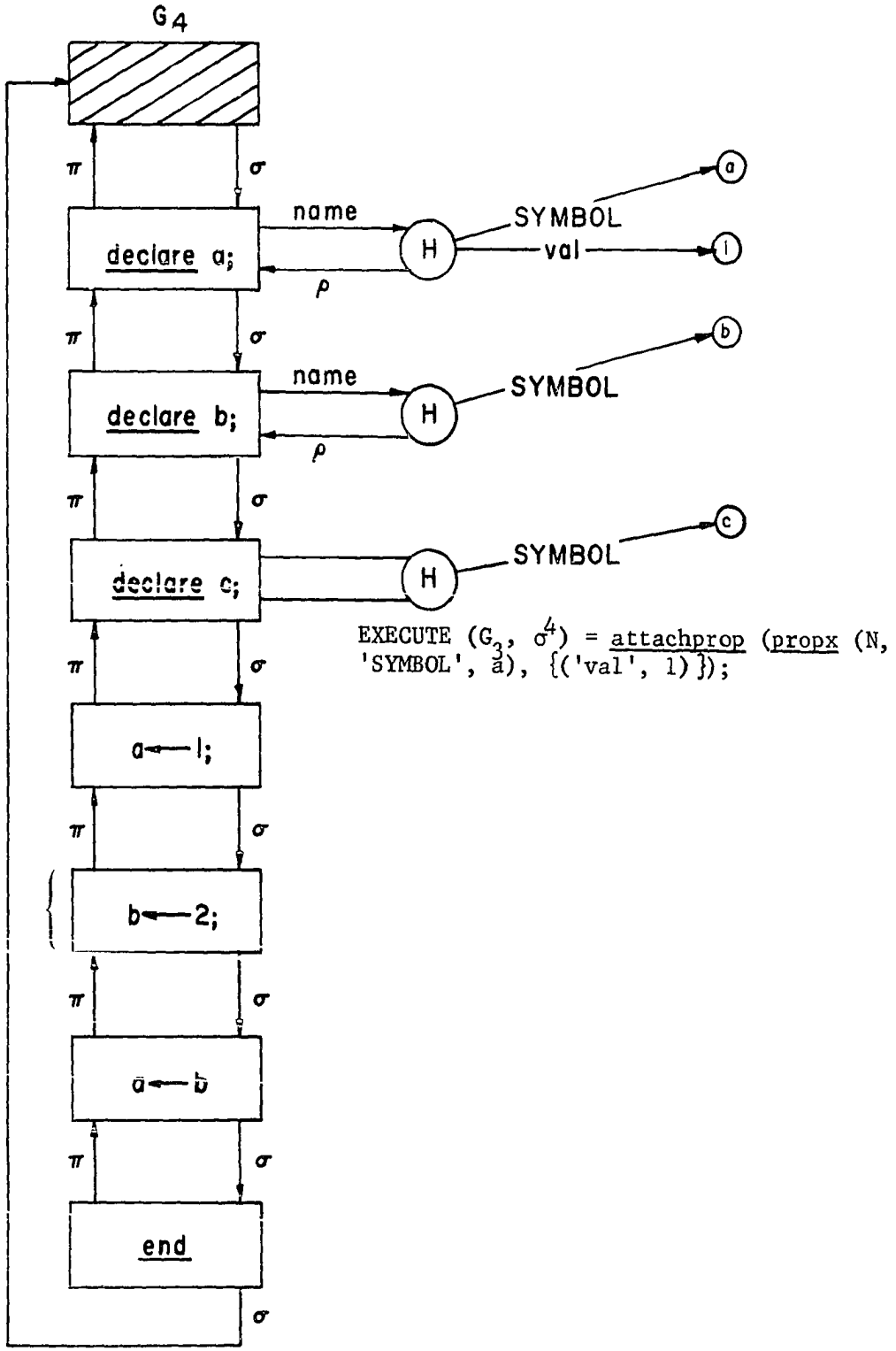
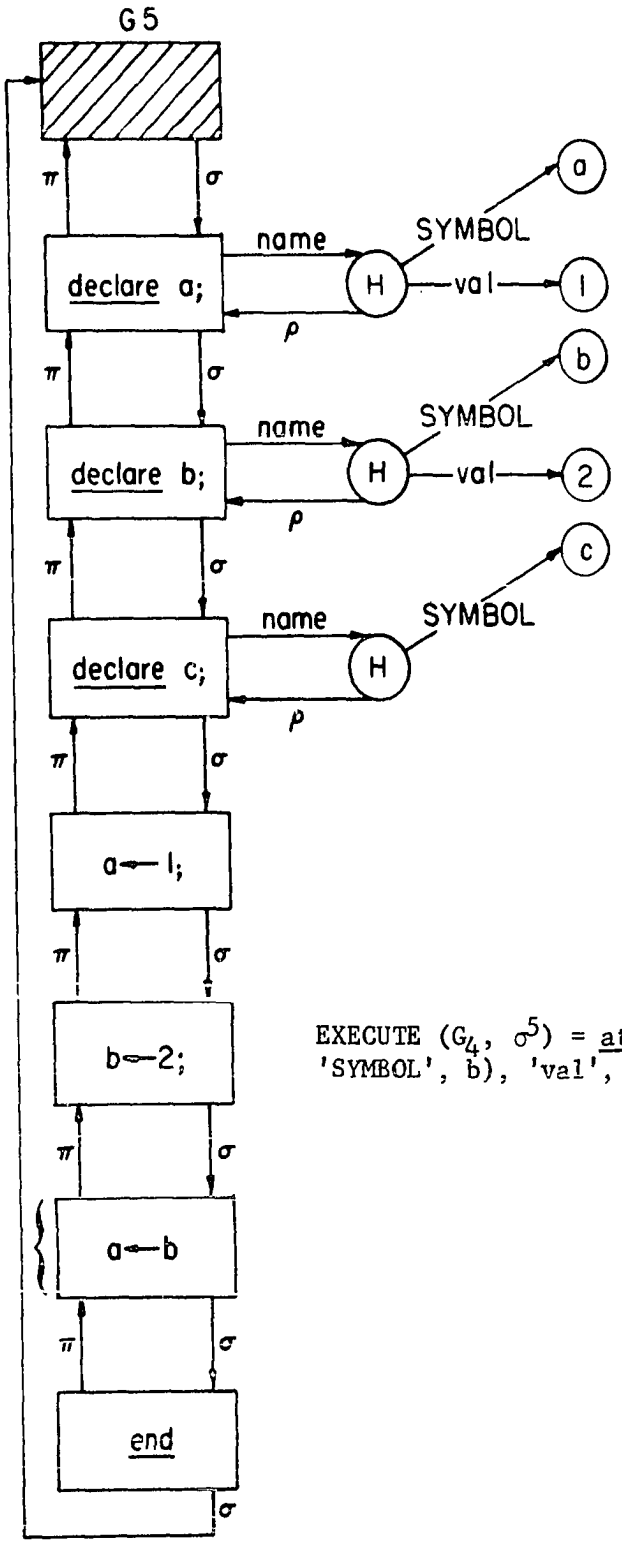
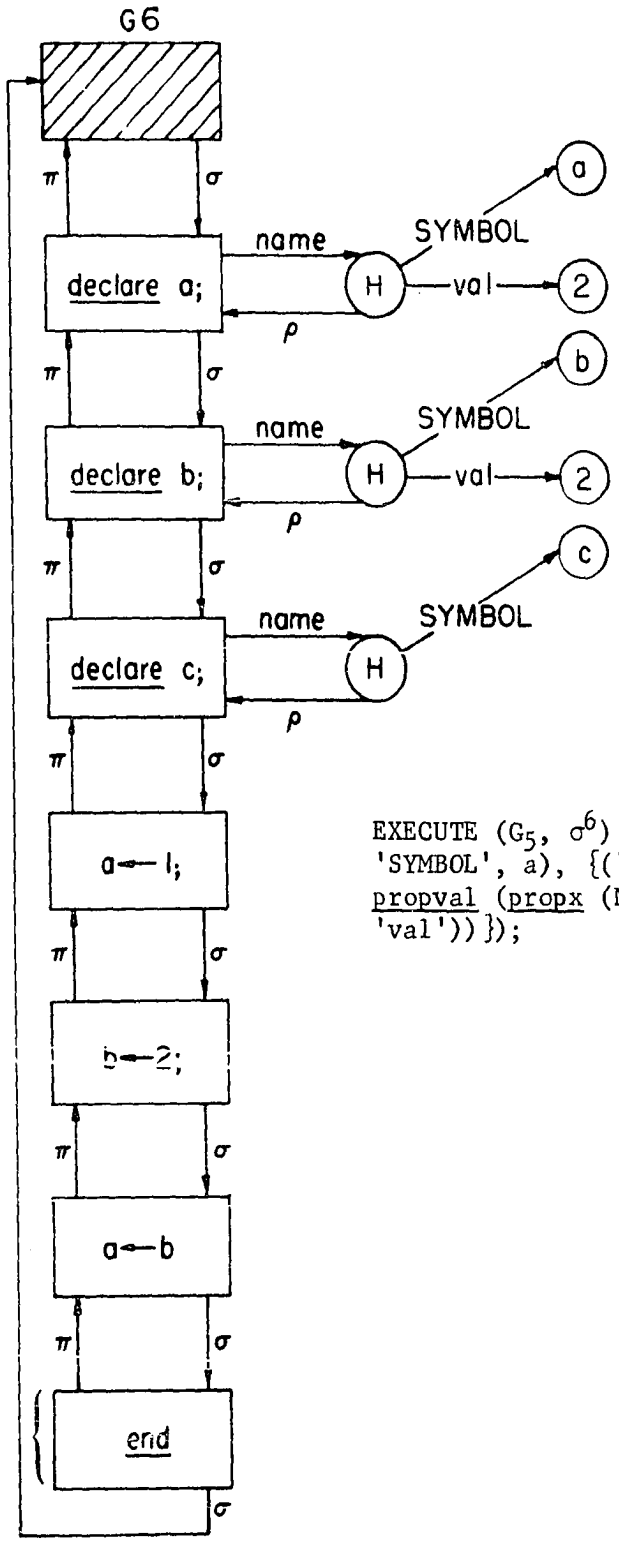


Figure 6e. Execution trace for example 5 with state = (G_4, σ^5)



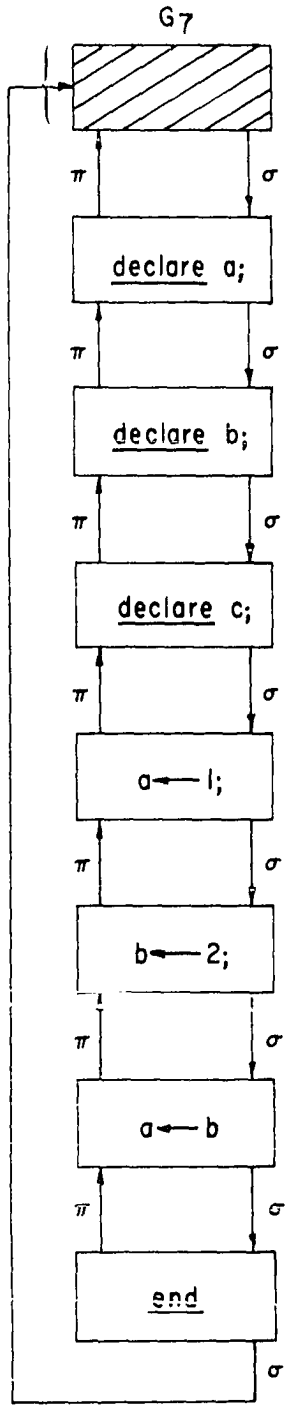
EXECUTE (G_4, σ^5) = attachprop (propx (N, 'SYMBOL', b), 'val', 2);

Figure 6f. Execution trace for example 5 with state = (G_5, σ^6)



EXECUTE (G_5, σ^6) = attachprop (propx (N, 'SYMBOL', a), {('val', propval (propx (N, 'SYMBOL', b), 'val'))});

Figure 6g. Execution trace for example 5 with state = (G_6, σ^7)



EXECUTE (G_6, σ^7) = destroy (N)

Figure 6h. Execution trace for example 5 with state = (G_7, σ^8)

Mini-Language 2 (ML-2)

Mini-Language 2 is a block structured language which employs a scoping rule not unlike that found in ALGOL 60. A program in ML-2 is defined to be a block which consists of the following: the key word begin followed by a blockhead consisting of a sequence of declarations of the form "declare identifier". The blockhead is followed by a blockbody consisting of a sequence of assignment statements and/or blocks. The blockbody is followed by the key word end. In ML-2 it is required that a given identifier be declared only once in a given blockhead. As in ML-1 an identifier used in a ML-2 program must be declared. If an identifier is used in a block and is not declared in that block's blockhead, then it is assumed to have the same meaning that it has in the textually enclosing block. A syntactic description of ML-2 is given in Figure 7.

P::=BLOCK

BLOCK::=begin D;S end

D::=D;δ|δ

δ::=declare identifier

S::=S;C|C

C::=α|BLOCK

α::=identifier ← term

term::=identifier|integer

Figure 7. Syntactic description of mini-language 2.

Simple names in ML-2, like those in ML-1, have an intrinsic 'SYMBOL' property whose value is its identifier representation. However, in ML-2, this property alone is not sufficient to resolve references. This is attributable to the fact that two names created via the execution of two distinct blockheads may possess the same 'SYMBOL' value. This degree of variability requires that simple names in ML-2 possess an additional intrinsic property that distinguishes between two creations of names with identical 'SYMBOL' properties in different blockheads. In addition, ML-2 names become nonexistent when the block in which they are declared is exited. To accommodate the correct resolution of names for referencing and destruction, the concept of a clock is introduced for the semantic description of ML-2 in AGM. The clock is treated as a special simple name with one intrinsic property, its 'SYMBOL' property. A special system identifier, $C, is used for the value of this property to avoid any conflict with user defined names. The clock name comes into existence prior to execution of a ML-2 program and is destroyed following execution. The execution of a begin statement has the effect of updating the 'val' property of the clock name ('val' is a secondary property), while the execution of an end statement destroys those names created in the block being exited.

The second intrinsic property associated with simple names in ML-2 is a 'TIME' property whose value is the value of the 'val' property of the clock at the time the name is created.

A program node in ML-2 possesses the same properties as in ML-1, i.e., a 'σ' property, a 'π' property, and possibly a 'name' property.

The translation of a ML-2 program into AGM form is slightly more complicated than for ML-2. Each begin, end, declaration, and assignment statement corresponds to a program node in the AGM representation as expected. However, the introduction of the clock requires the use of two special nodes, an initial node, called an IC node, and a final node, called a DC node. The IC node must precede the first begin node and is used to create and initialize the special clock name. The DC node must follow the last end node and is used to destroy the clock.

Following translation, execution of the AGM representation proceeds as follows.

IC node

State prior to execution = (G_0, σ),

where the contents of σ = "IC".

NAL code:

attachprop (σ, 'name', createname ({ ('SYMBOL', $C)}));

attachprop (propx (N, 'SYMBOL', $C), {('$\rho$', σ), ('val', 0)});

New state = (G_1, σ^2), where G_1 differs from G_0 in that σ has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "$C", a 'val' property with value "0", and a 'ρ' property with value σ.

Begin node

State prior to exeuction = (G_1, σ^j),

where the contents of σ^j = "begin".

NAL code:

attachprop (propx (N, 'SYMBOL', $C), {('val',

propval (propx (N, 'SYMBOL', $C), 'val') + 1)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that the 'val' property of the special clock name has been incremented by 1.

Declaration node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "declare idenfitifer;".

NAL code:

attachprop (σ^j, 'name', createname ({('SYMBOL', identifier), ('TIME',

propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', identifier), 'TIME',

Max), {('ρ', σ^j)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of the clock, and a 'ρ' property with value σ^j.

Assignment node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "identifier1 ← identifier2;".

NAL code:

attachprop (propx (propset (N, 'SYMBOL', identifier1),
'TIME', Max), {('val', propval (propx (propset (N,
'SYMBOL', identifier2), 'TIME', Max), 'val'))});

The situation where the right hand side is a constant, n, would be handled as in ML-1 with the following NAL code:

attachprop (propx (propset (N, 'SYMBOL', identifier),
'TIME', Max, {('val', n)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that the 'val' property of name k now has value v where k and v are defined as follows: Given all names in N with 'SYMBOL' property "identifier1", k is that one having the largest 'TIME' property value. Given all names in N with 'SYMBOL' property "identifier2", v is the value of the 'val' property for that one having the largest 'TIME' property value (or v is simply n in the case of the simplified assignment form).

End node

State prior to execution = (G_i, σ^j),
where the contents of σ^j = "end;".

NAL code:

destroy (propset (N, 'TIME', Max);

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that all names declared in the exited block are destroyed.

DC node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "DC".

NAL code:

destroy (propset (N, 'SYMBOL', $C));

New state = (G_{i+1}, σ^j), where G_{i+1} differs from G_i in that the name with 'SYMBOL' property "$C" is destroyed.

Example 6

ML-2 Program

```
begin
    declare x;
    declare y;
    x:=1;
    y:=2;
    begin
        declare y;
        y:=x
    end;
    begin
        declare x;
        x:=y
    end
end
```

The ML-2 program listed above illustrates the use of default scoping in accessing names. A partial trace of this program is given in Figures 8a-8f. The actual state transformations are effected by NAL code as indicated below.

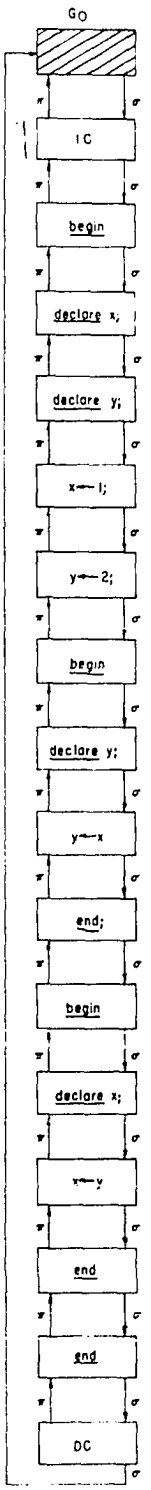


Figure 8a. Initial AGM representation for example 6 with state = (G_0, σ)

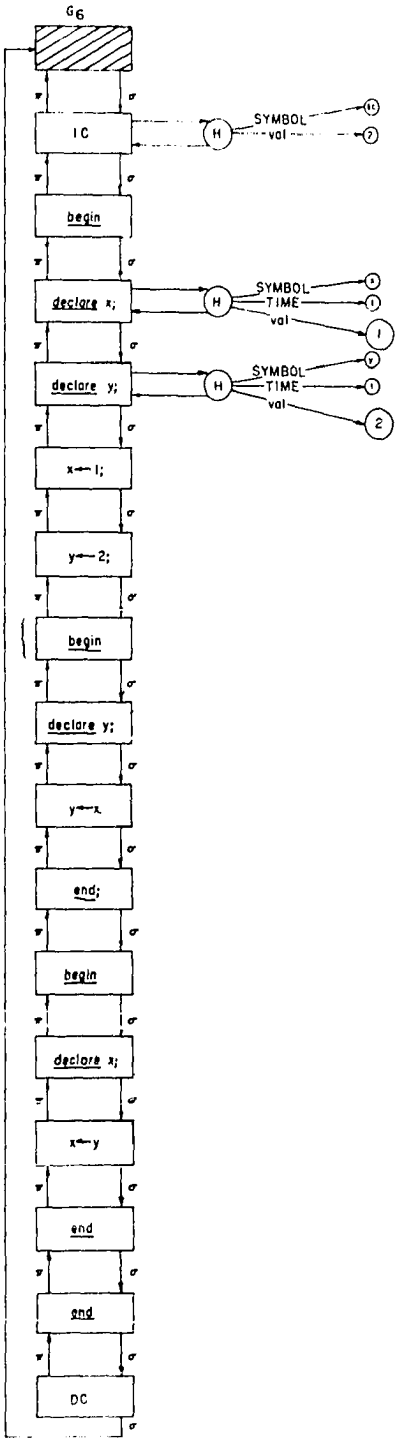


Figure 8b. Execution trace for example 6 with state = (G_6, σ)

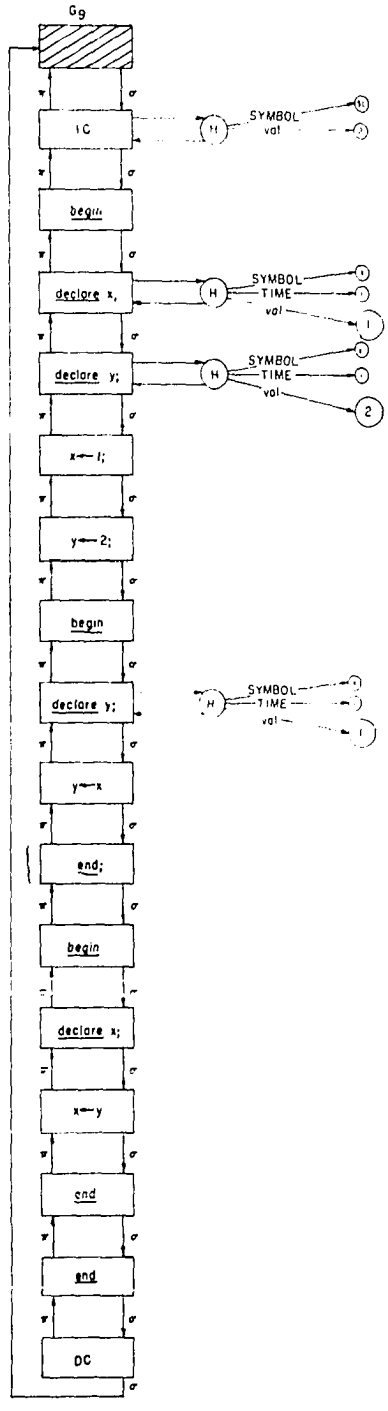


Figure 8c. Execution trace for example 6 with state = (G_9, σ^{10})

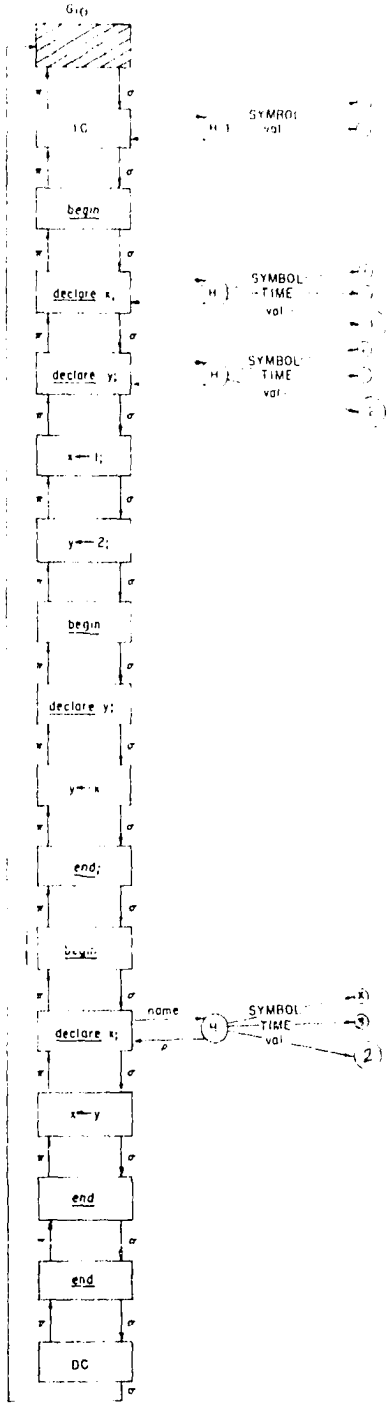


Figure 8d. Execution trace for example 6 with state = (G_{10}, σ^{11})

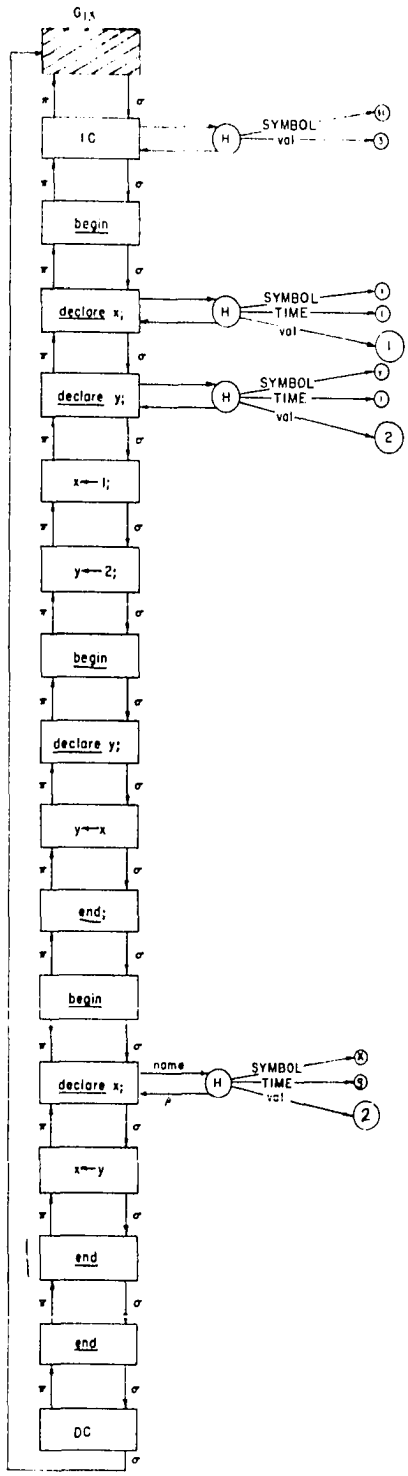


Figure 8e. Execution trace for example 6 with state = (G_{13}, σ^{14})

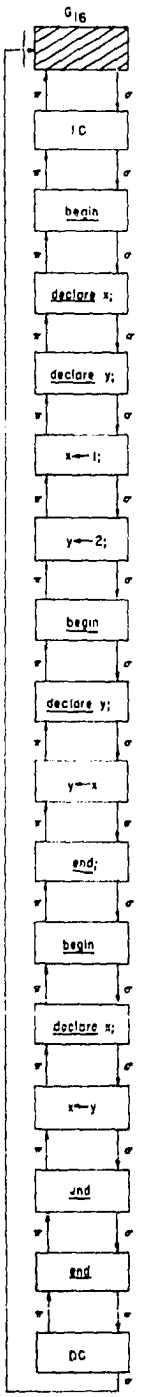


Figure 8f. Execution trace for example 6 with state = (G_{16}, σ^{17})

EXECUTE (G_0, σ) =

attachprop (σ, 'name', createname ({('SYMBOL', $C)}));

attachprop (propx (N, 'SYMBOL', $C), {('$\rho$', σ), ('val', 0)});

EXECUTE (G_1, σ^2) =

attachprop (propx (N, 'SYMBOL', $C), {('val',
propval (propx (N, 'SYMBOL', $C), 'val')+1)});

EXECUTE (G_2, σ^3) =

attachprop (σ^3, 'name', createname ({('SYMBOL', x), ('TIME',
propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', x),
'TIME', Max), {('ρ', σ^3)});

EXECUTE (G_3, σ^4) =

attachprop (σ^4, 'name', createname ({('SYMBOL', y), ('TIME',
propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', y), 'TIME',
Max), {('ρ', σ^4)});

EXECUTE (G_4, σ^5) =

attachprop (propx (propset (N, 'SYMBOL', x), 'TIME',
Max), {('val', 1)});

EXECUTE (G_5, σ^6) =

attachprop (propx (propset (N, 'SYMBOL', y), 'TIME',
Max), {('val', 2)});

EXECUTE (G_6, σ^7) =

attachprop (propx (N, 'SYMBOL', $C), {('val', propval (propx (N, 'SYMBOL', $C), 'val')+1)});

EXECUTE (G_7, σ^8) =

attachprop (σ^8, 'name', createname ({('SYMBOL', x), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));
attachprop (propx (propset (N, 'SYMBOL', x), 'TIME', Max), {('ρ', σ^8)});

EXECUTE (G_8, σ^9) =

attachprop (propx (propset (N, 'SYMBOL', y), 'TIME', Max), {('val', propval (propx (propset (N, 'SYMBOL', x), 'TIME', Max), 'val'))});

EXECUTE (G_9, σ^{10}) =

destroy (propset (N, 'TIME', Max));

EXECUTE (G_{10}, σ^{11}) =

attachprop (propx (N, 'SYMBOL', $C), {('val', propval (propx (N, 'SYMBOL', $C), 'val')+1)});

EXECUTE (G_{11}, σ^{12}) =

attachprop (σ^{12}, 'name', createname ({('SYMBOL', x), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));
attachprop (propx (propset (N, 'SYMBOL', x), 'TIME', Max), {('ρ', σ^{12})});

EXECUTE (G_{12}, σ^{13}) =

attachprop (propx (propset (N, 'SYMBOL', x), 'TIME',
Max), {('val', propval (propx (propset (N, 'SYMBOL', y),
'TIME', Max), 'val'))});

EXECUTE G_{13}, σ^{14}) =

destroy (propset (N, 'TIME', Max));

EXECUTE (G_{14}, σ^{15}) =

destroy (propset (N, 'TIME', Max));

EXECUTE (G_{15}, σ^{16}) =

destroy (propset (N, 'SYMBOL', $C));

Mini-Language 3 (ML-3)

Mini-Language 3 is based on a block structured concept introduced by George and Sager (23). The main difference between ML-2 and ML-3 is the degree of variability allowed in ML-3. In ML-3, there is no default scoping rule employed, i.e., all identifiers used in a blockbody must be declared in that block's blockhead. A declaration in ML-3 can take on one of the following forms:

1) new identifier;

This declaration creates a new name whose scope is the block in which it is declared, exclusive of all textually enclosed blocks. However, the scope of this name may be extended inward via the application of certain other declarations.

2) local identifier;

This declaration is similar to the "new identifier;" declaration, except that local identifiers may not have their scopes extended.

3) near identifier;

This declaration has as its effect the extension of the scope of the most-recent name created by a "new identifier;" declaration inward to include the block where "near identifier;" occurs.

4) far identifier;

This declaration behaves as the "near identifier;" declaration, but it extends the scope of the oldest declaration of a "new identifier;".

A syntactic description of ML-3 is given in Figure 9.

P::=BLOCK

BLOCK::=begin D;S end

D::=D;δ |δ

δ::=new identifier |local identifier |near identifier |far identifier

S::=S;C |C

C::=α |BLOCK

α::=identifier ← term

term::=identifier |integer

Figure 9. Syntactic description of mini-language 3.

One of the more interesting features of ML-3 is that it permits the creation of scoping "holes". Consider, for example, the following ML-3 program.

```
begin                                   BLOCK #1
    new x;
    x ← 1;
    begin                               BLOCK #2
        new y;
        y ← 2;
        begin                           BLOCK #3
            near x;
            x ← 3
        end
    end
end
```

The effect of the "near x" in BLOCK #3 is to extend the scope of the x declared in BLOCK #1 to include BLOCK #3, but not BLOCK #2. Thus, BLOCK #2 represents a hole in the scope of x. In effect, the existence of scoping holes requires the use of sharing, defined over certain boundaries. The declaration of "near x;" thus creates a new name which shares certain properties with the x in BLOCK #1. The exiting of BLOCK #3 destroys this sharing arrangement.

A simple name in ML-3 has two intrinsic properties, a 'SYMBOL' property whose value is an identifier and a 'TIME' property whose value is the current value of the 'val' property of the system clock when the name is created. (The system clock is the same as the one defined for ML-2.) In addition to these properties, a simple name in ML-3 possesses a 'ρ' property, as previously defined, and 'loc' property whose value is a subname. The subname itself possesses a 'share' property and a 'cont' property. The value of the 'share' property is either Y or P. If the value is Y, this indicates that this subname may also be used as a subname of another name with a 'loc' property, i.e., it may share.

Program nodes in ML-3 possess the same properties as before: a 'σ'

property, a 'π' property, and possible a 'name' property.

The translation of a ML-3 program into AGM form again requires the special IC and DC nodes to precede and follow the program text respectively. Aside from this, translation is one-one with each begin, end, assignment and declaration statement forming a program node in the AGM representation as in ML-2.

Execution is then defined as follows:

IC node

State prior to execution = (G_0, σ),

where the contents of σ = "IC".

NAL code:

attachprop (σ, 'name', createname ({('SYMBOL', $C)}));

attachprop (propx (N, 'SYMBOL', $C), {('$\rho$', σ), ('val', 0)});

New state = (G_1, σ^2), where G_1 differs from G_0 in that σ has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value $C, a 'val' property with value "0", and a 'ρ' property with value σ.

Begin node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "begin".

NAL code:

attachprop (propx (N, 'SYMBOL', $C), {('val',

propval (propx (N, 'SYMBOL', $C), 'val')+1)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that the 'val' property of the special clock name has been incremented by 1.

Declaration nodes

Case I.

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "new identifier;".

NAL code:

attachprop (σ^j, 'name', createname ({('SYMBOL', identifier), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', identifier), {('ρ', σ^j), ('loc', createname ({('cont', Ω), ('share', Y)}))});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of the clock, a 'ρ' property whose value is σ^j, and a 'loc' property whose value is a newly created subname. This subname has a 'cont' property whose value is undefined, (Ω), and a 'share' property whose value is Y, i.e., this subname may be shared.

Case II.

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "local identifier;".

NAL code:

attachprop (σ^j, 'name', createname ({('SYMBOL', identifier), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', identifier), {('ρ', σ^j), ('loc', createname ({('cont', Ω), ('share', P)}))});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of the clock, a 'ρ' property whose value is σ^j and a 'loc' property whose value is a newly created subname. This subname has a 'cont' property whose value is undefined, (Ω), and a 'share' property whose value is P, i.e., this subname may not be shared

Case III.

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "near identifier;".

NAL code:

attachprop (σ^j, 'name', createname ({('SYMBOL', identifier), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', identifier), {('ρ', σ^j), ('loc', propval (propx (propset (propset (N, 'SYMBOL, identifier), 'loc'.'share', Y), 'TIME', Min), 'loc'))});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of the clock, a 'ρ' property whose value is σ^j and a 'loc' property whose value is the 'loc' value of that name which in turn has a 'SYMBOL' property "identifier", a sharable subname, and a Max 'TIME' value. The result of this declaration is the sharing of this 'loc' value between these names.

Case IV.

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "far identifier;".

NAL code:

```
attachprop (σ^j, 'name', createname ({('SYMBOL', identifier), ('TIME',
        propval (propx (N, 'SYMBOL', $C), 'val'))}));
attachprop (propx (propset (N, 'TIME', propval (propx (N,
        'SYMBOL', $C), 'val')), 'SYMBOL', identifier), {('ρ',
        σ^j), ('loc', propval (propx (propset (propset (N,
        'SYMBOL', identifier), 'loc'.'share', Y), 'TIME',
        Min), 'loc'))});
```

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node. This header node has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of

the clock, a 'ρ' property whose value is σ^j and a 'loc' property whose value is the 'loc' value of that name which in turn has a 'SYMBOL' property "identifier", a sharable subname and a Min 'TIME' value. The result of this declaration is the sharing of this 'loc' value between the names.

A few remarks are in order concerning the resolution of the names that a near or far declaration refers to. As defined, this resolution is accomplished as follows:

1) Select that subset of N whose 'SYMBOL' property is identifier, i.e., $X \subseteq N$ where

   X = propset (N, 'SYMBOL', identifier).

2) From X, choose those elements with sharable subnames, i.e., $X' \subseteq X$ where

   X' = propset (X, 'loc'.'share', Y),

   or in expanded notation,

   $X' = \{Z \mid Z \in X \wedge$ propval (propval (Z, 'loc'), 'share') = Y$\}$.

3) From X', choose that element with the largest (near) or smallest (far) 'TIME' value, i.e.,

   X" = propx (X', 'TIME', Max)

   or

   X" = propx (X', 'TIME', Min).

The added overhead of the double property selector in step 2 could be avoided if the 'share' property was associated with the name rather than the subname. If this were the case then step 2 could be written as

X' = propset (X, 'share', Y).

Such an optimization could be incorporated in ML-3 with no loss of generality. The variability gained by making 'share' a property of a subname would become an issue in a language in which names could possess several subnames. In this case it would be possible to let some subnames share while others could not.

Assignment node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "identifier1 ← identifier2;".

NAL code:

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', identifier1), 'loc'), {('cont', propval (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', identifier2), 'loc'), 'cont'))});

For the case "identifier1 ← n;" the NAL code is:

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', identifier1), 'loc'), {('cont', n)});

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that the 'cont' property of the subname of the name with 'SYMBOL' property "identifier1" and 'TIME' property equal to the current value of the clock has been set to agree with the value of the 'cont' property of the subname of the appropriately chosen name for identifier2 (or the number n for the simpler case).

Name accessing in ML-3 is made even more interesting by the fact that it may be performed in two different ways. As defined above, a reference occurrence of an identifier, X, is accommodated by

propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', X).

In other words, the universe N is interrogated to determine all names declared in the current block and then that name with 'SYMBOL' property X is selected from this set.

An alternative form for name accessing in ML-3 is

propx (propset (N, 'SYMBOL', X), 'TIME', propval (propx (N, 'SYMBOL', $C), 'val'));.

In this form all names with 'SYMBOL' property X are selected and then that name declared in the current block is chosen.

From an efficiency point of view, the most efficient form of name accessing is program dependent. It would be possible to determine at translation time which form would optimize accessing and then "tune" the name accessing function to the program.

End node

State prior to execution = (G_i, σ^j), where the contents of σ^j = "end;".

NAL code:

destroy (propset (N, 'TIME', Max));

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that all names declared in the exited block are destroyed.

DC node

State prior to execution = (G_i, σ^j),

where the contents of σ^j = "DC".

NAL code:

```
destroy (propset (N, 'SYMBOL', $C));
```

New state = (G_{i+1}, σ^{j+1}), where G_{i+1} differs from G_i in that the name with 'SYMBOL' property "$C" is destroyed.

Example 7

ML-3 Program

```
begin
    new a;
    new b;
    a ← 1;
    b ← 2;
    begin
        new a:
        local b;
        a ← 3;
        b ← 4;
        begin
            near b;
            far a;
            a ← 5;
            b ← a
        end
    end
end
```

A partial execution trace of the ML-3 program given above is given in Figures 10a-10e. The NAL code effecting the actual state

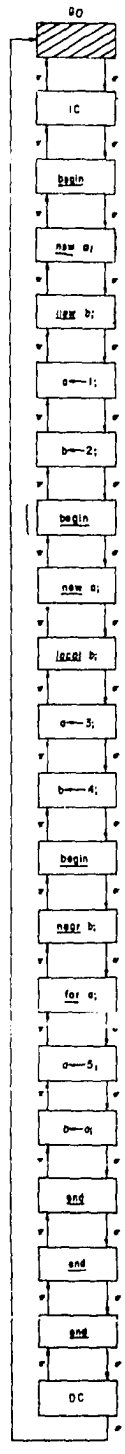


Figure 10a. Initial AGM representation for example 7 with state = (G_0, σ)

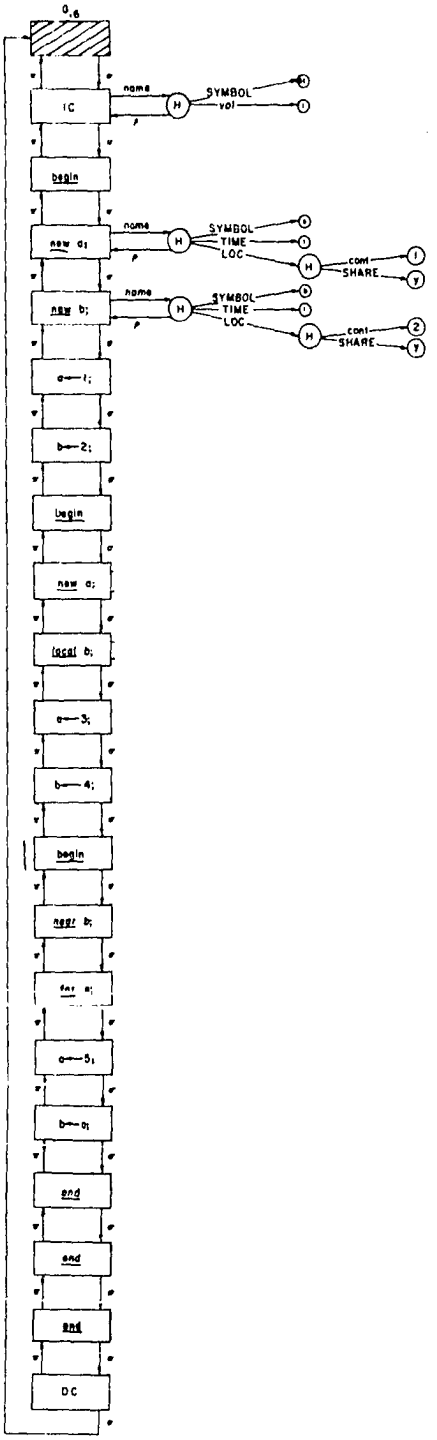


Figure 10b. Execution trace for example 7 with state = (G_6, σ^7)

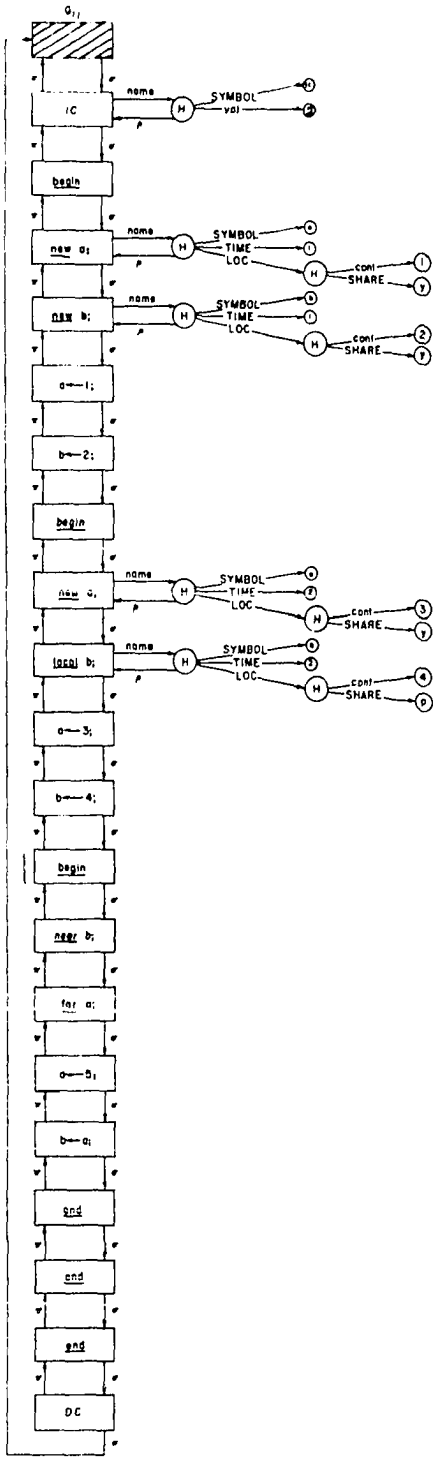


Figure 10c. Execution trace for example 7 with state = (G_{11}, σ^{12})

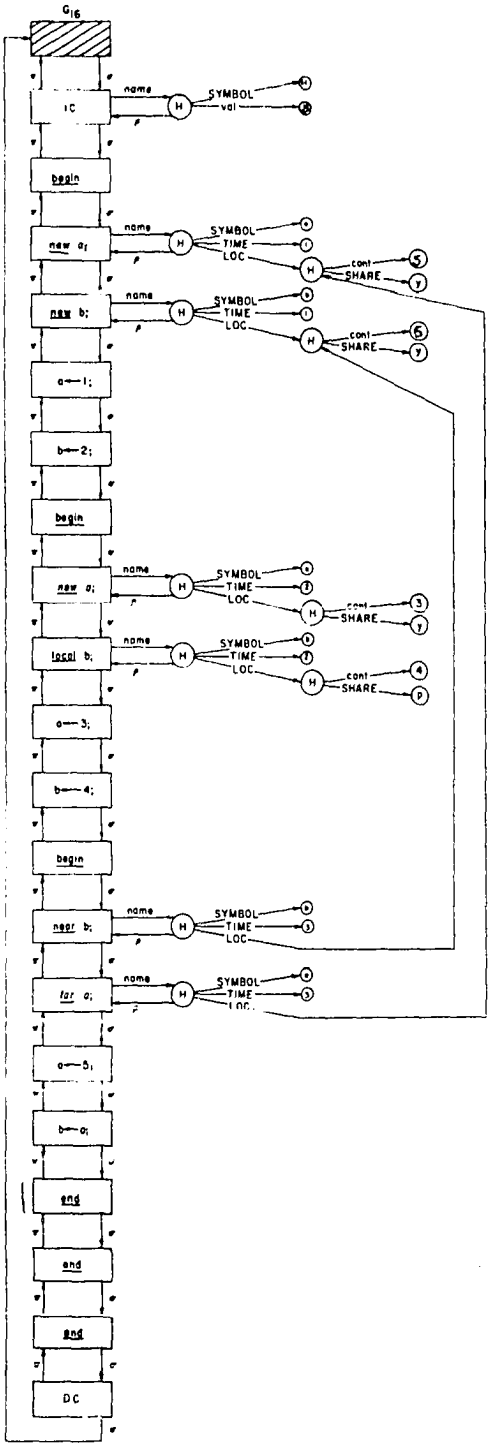


Figure 10d. Execution trace for example 7 with state = (G_{12}, σ^{17})

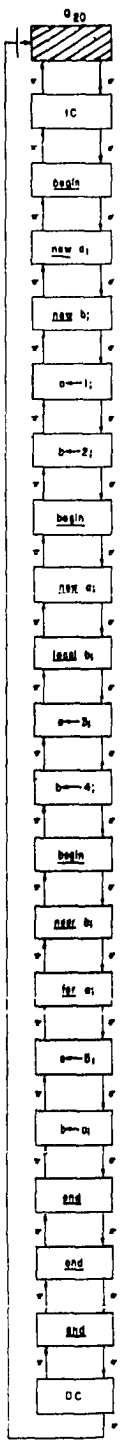


Figure 10e. Execution trace for example 7 with state = (G_{20}, σ^{21})

transformations is indicated below.

EXECUTE (G_0, σ) =

attachprop (σ, 'name', createname ({('SYMBOL', $C)}));

attachprop (propx (N, 'SYMBOL', $C), {('$\rho$', σ), ('val', 0)});

EXECUTE (G_1, σ^2) =

attachprop (propx (N, 'SYMBOL', $C), {('val', propval (propx (N, 'SYMBOL', $C), 'val')+1)});

EXECUTE (G_2, σ^3) =

attachprop (σ^3, 'name', createname ({('SYMBOL', a), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', a), {('ρ', σ^3), ('loc', createname ({('cont', Ω), ('share', Y)}))});

EXECUTE (G_3, σ^4) =

attachprop (σ^4, 'name', createname ({('SYMBOL', b), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', b), { ('ρ', σ^4), ('loc', createname ({('cont', Ω), ('share', Y)}))});

EXECUTE (G_4, σ^5) =

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', a), 'loc'), {('cont', 1)});

EXECUTE (G_5, σ^6) =

attachprop (propval (propx (propset (N, 'TIME',
propval (propx (N, 'SYMBOL', $C), 'val')),
'SYMBOL', b), 'loc'), {('cont', 2)});

EXECUTE (G_6, σ^7) =

attachprop (propx (N, 'SYMBOL', $C), { ('val', propval (propx (N,
'SYMBOL', $C), 'val')+1)});

EXECUTE (G_7, σ^8) =

attachprop (σ^8, 'name', createname ({('SYMBOL', a), ('TIME',
propval (propx (N, 'SYMBOL', $C), 'val'))}));
attachprop (propx (propset (N, 'TIME', propval (propx (N,
'SYMBOL', $C), 'val')), 'SYMBOL', a), {('ρ', σ^8), ('loc',
createname ({('cont', Ω), ('share', Y)}))});

EXECUTE (G_8, σ^9) =

attachprop (σ^9, 'name', createname ({('SYMBOL', b), ('TIME',
propval (propx (N, 'SYMBOL', $C), 'val'))}));
attachprop (propx (propset (N, 'TIME', propval (propx (N,
'SYMBOL', $C), 'val')), 'SYMBOL', b), {('ρ', σ^9), ('loc',
createname ({('cont', Ω), ('share', P)}))});

EXECUTE (G_9, σ^{10}) =

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N,
'SYMBOL', $C), 'val')), 'SYMBOL', a), 'loc'), {('cont',
3)});

EXECUTE (G_{10}, σ^{11}) =

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', b), 'loc'), {('cont', 4)});

EXECUTE (G_{11}, σ^{12}) =

attachprop (propx (N, 'SYMBOL', $C), {('val', propval (propx (N, 'SYMBOL', $C), 'val')+1)});

EXECUTE (G_{12}, σ^{13}) =

attachprop (σ^{13}, 'name', createname ({ ('SYMBOL', b), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', b), {('ρ', σ^{13}), ('loc', propval (propx (propset (propset (N, 'SYMBOL', b), 'loc'.'share', Y), 'TIME', Max), 'loc'))});

EXECUTE (G_{13}, σ^{14}) =

attachprop (σ^{14}, 'name', createname ({('SYMBOL', a), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', a), {('ρ', σ^{14}), ('loc', propval (propx (propset (propset (N, 'SYMBOL', a), 'loc'.'share', Y), 'TIME', Min), 'loc'))});

EXECUTE (G_{14}, σ^{15}) =

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', a), 'loc'), {('cont', 5)});

EXECUTE (G_{15}, σ^{16}) =

attachprop (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', b), 'loc'), {('cont', propval (propval (propx (propset (N, 'TIME', propval (propx (N, 'SYMBOL', $C), 'val')), 'SYMBOL', a), 'loc'), 'cont'))});

EXECUTE (G_{16}, σ^{17}) =

destroy (propset (N, 'TIME', Max));

EXECUTE (G_{17}, σ^{18}) =

destroy (propset (N, 'TIME', Max));

EXECUTE (G_{18}, σ^{19}) =

destroy (propset (N, 'TIME', Max));

EXECUTE (G_{19}, σ^{12}

destroy (propset (N, 'SYMBOL', $C));

Mini-Language 4 (ML-4)

In Mini-Language 4 some of the closure mechanism that can be used in binding the nonlocal variables of procedures are considered. In general, procedures have two types of nonlocal variables, parameters and globals. In both cases, the closure of a nonlocal requires the binding of certain properties to the nonlocal. The values bound to these properties are extracted from names that exist external to the procedure. Since the binding strategies for both parameters and globals are quite similar, only globals will be considered in ML-4. In particular, two types of explicit closure mechanisms are supported in ML-4, a value closure and a share closure. These mechanisms are based on the closure schemes available in EL1 (74).

The most restrictive form of closure in ML-4 is the value closure. This closure has the form "(identifier value term)" where the identifier to the left of value is the nonlocal being closed. The effect of this closure is to bind this nonlocal to the value of the term.

The share closure has the form "(identifier1 share identifier2)" and represents a more general form of closure. The effect of this closure is to bind identifier1 to the location (or subname) property of identifier2.

In addition to these two explicit forms of closure, ML-4 also supports a default closure. Under the default closure, an unclosed nonlocal appearing in a procedure is resolved to refer to the most-recent dynamically created instance of that name.

As noted in the syntactic description of ML-4 given in Figure 11, ML-4 supports two types of simple names, procedure names and simple value names. These names possess the same intrinsic properties; a 'SYMBOL' property whose value is an identifier and a 'TIME' property whose value is the value of the 'val' property of the system clock (as previously defined) at the time the name is created. In addition to these properties, both types of names possess a 'ρ' property as discussed in the previous languages. Simple names also possess a 'loc' property whose value is a subname. The subname itself possesses a 'cont' property, whose value is either an integer, for simple value names, or a translated AGM representation of a procedure body, for procedure names.

```
PROGRAM::=D;S end
D::=D;δ |δ
δ::=new identifier |procvar identifier
S::=S;C |C
C::=α|β
α::=identifier ← term |call identifier
term::=identifier |integer
β::=identifier = procval
procval::=identifier |proctext |closure
proctext::=[[local-list; pstate-list]]
local-list::=local-list; new identifier |new identifier
pstate-list::=pstate-list; α|α
closure::=[identifier, close {clist}]
clist::=clist;K |K
K::=(identifier share identifier) |(identifier val term)
```

Figure 11. Syntactic description of mini-language 4.

Program nodes are handled somewhat differently in ML-4 due to the fact that procedure invocation results in an interruption of normal flow. Program nodes normally possess a 'σn' property for normal flow of

execution, a 'π' property which determines the universe N, and possibly a 'name' property as defined before. In ML-4, however, program nodes may acquire a 'σB' property which determines a branch to a procedural text.

Translation of ML-4 programs follows the normal one-one pattern of statements to program nodes with the DC and IC nodes included for the system clock as before. Execution of ML-4 programs is then defined as follows, where Ψ = some sequence of σn's and σB's.

IC node

State prior to execution = $(G_0, \sigma n)$,

where the contents of σn = "IC".

NAL code:

attachprop (σn, 'name', createname ({('SYMBOL', \$C)}));

attachprop (propx (N, 'SYMBOL', \$C), {('ρ', σn), ('val', 0)});

New state = $(G_1, \sigma n^2)$, where G_1 differs from G_0 in that σn has acquired a 'name' property whose value is a new header node for a name. This name has a 'SYMBOL' property with value \$C, a 'val' property with value 0, and a 'ρ' property with value σn.

Declaration nodes

Case I.

State prior to execution = $(G_i, \sigma n^j)$,

where the contents of σn^j = "procvar identifier;".

NAL code:

attachprop (σn^j, 'name', createname ({('SYMBOL',
identifier), ('TIME', propval (propx (N, 'SYMBOL', $C),
'val'))}));

attachprop (propx (propset (N, 'SYMBOL', identifier), 'TIME',
Max), {('ρ', σn^j), ('loc', createname ({('cont',
B)}))});

(B represents a blank program node, the precontext node for the procedure body.)

New state = (G_{i+1}, σn^{j+1}), where G_{i+1} differs from G_i in that σ^j has acquired a 'name' property whose value is a new header node for a name. This name has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of the clock, a 'ρ' property whose value is σn^j, and a 'loc' property whose value is a newly created subname. This subname has a 'cont' property whose value is an empty program node, B.

Case II.

State prior to execution = (G_i, Ψ),

where the contents of Ψ = "new identifier;".

NAL code:

attachprop (Ψ, 'name', createname ({('SYMBOL', identifier), ('TIME',
propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', identifier), 'TIME',
Max), {('ρ', Ψ), ('loc', createname ({('cont', Ω)}))});

New state = (G_{i+1}, Ψon), where G_{i+1}, differs from G_i in that Ψ has acquired a 'name' property whose value is a new header node for a name. This name has a 'SYMBOL' property with value "identifier", a 'TIME' property whose value is the current value of the 'val' property of the clock, a 'ρ' property whose value is a newly created subname. This subname has a 'cont' property whose value is undefined (Ω).

Assignment node

State prior to execution = (G_i, Ψ),
where the contents of Ψ = "identifier1 ← identifier2;".

NAL code:

attachprop (propval (propx (propset (N, 'SYMBOL', identifier1),
'TIME', Max), 'loc'), {('cont', propval (propval (
propx (propset (N, 'SYMBOL', identifier2), 'TIME',
Max), 'loc'), 'cont'))});

If the left-hand side is a constant n, the NAL code would be

attachprop (propval (propx (propset (N, 'SYMBOL', identifier1),
'TIME', Max), 'loc'), {('cont', n)});

New state = (G_{i+1}, Ψon), where G_{i+1} differs from G_i in that the 'cont' property of the subname of the name with 'SYMBOL' property "identifier1" and Max value on the 'TIME' property has been updated

to agree with the value of the 'cont' property of the subname of identifier2 (or the number n for the simpler case).

Procedure assignment nodes

Case I.

State prior to execution = $(G_i, \sigma n^j)$,

where the contents of σn^j = "identifier1 = identifier2;".

NAL code:

```
attachprop (propval (propx (propset (N, 'SYMBOL',
            identifier1), 'TIME, Max), 'loc'), {('cont',
            COPY (propval (propval (propx (propset (N, 'SYMBOL',
            identifier2), 'TIME', Max), 'loc'), 'cont'))) });
```

COPY is simply a macro facility of NAL which makes a copy of the subgraph which is its argument and returns this copy. In copying a procedure, however, care must be taken to preserve any share closures which have been created for the procedure's nonlocals. Consider, for example, the partial graph pictured on the following page. The name with 'SYMBOL' property "b" has a 'loc' property whose value is the 'loc' property of some other name. In a situation such as this, COPY must not duplicate either the 'loc' value or the 'cont' value. It must simply copy the 'loc' edge. Such situations are decided on the basis of the indegree of the 'loc' values.

New state = $(G_{i+1}, \sigma n^{j+1})$, where G_{i+1} differs from G_i in that the 'cont' property of the subname of the name with 'SYMBOL' property

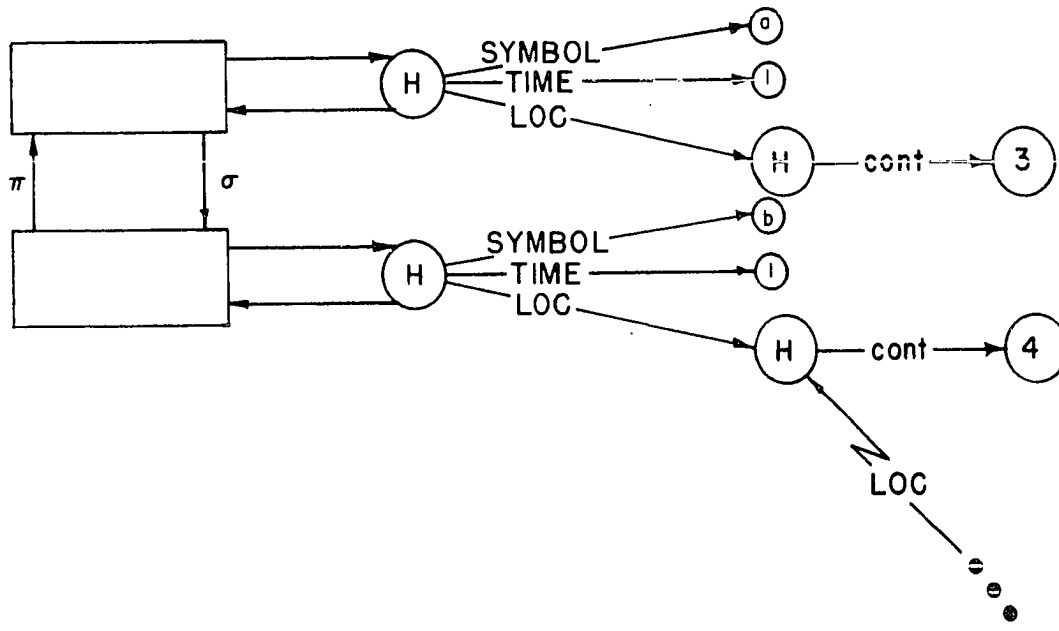


Diagram 1. Preservation of share closures

"identifier1" and Max value on the 'TIME' property has acquired a copy of the value of the 'cont' property of the subname of identifier2.

Case II.

State prior to execution = $(G_i, \sigma n^j)$,

where the contents of σn^j = "identifier = $[\![S_1; S_2; \ldots; S_n]\!]$;".

NAL code:

attachprop (propval (propval (propx (propset (N, 'SYMBOL',
identifier1), 'TIME', Max), 'loc'),
'cont'), {('σn', H(X))});

attachprop (H(X), {('π', propval (propval (propx (propset (N,
'SYMBOL', identifier1), 'TIME', Max), 'loc'),
'cont'))});

attachprop (T(X), {('σn', propval (propval (propx (propset (N,
'SYMBOL', identifier1), 'TIME', Max), 'loc'),
'cont'))});

where X = TRANSLATE $[\![S_1; S_2; \ldots; S_n]\!]$

and H(X) is the first program node of X

and T(X) is the last program node of X.

The pictorial representation of this translation process is given on the following page.

New state = $(G_{i+1}, \sigma n^{j+1})$, where G_{i+1} differs from G_i in that the value of the 'cont' property of the subname of the name with 'SYMBOL'

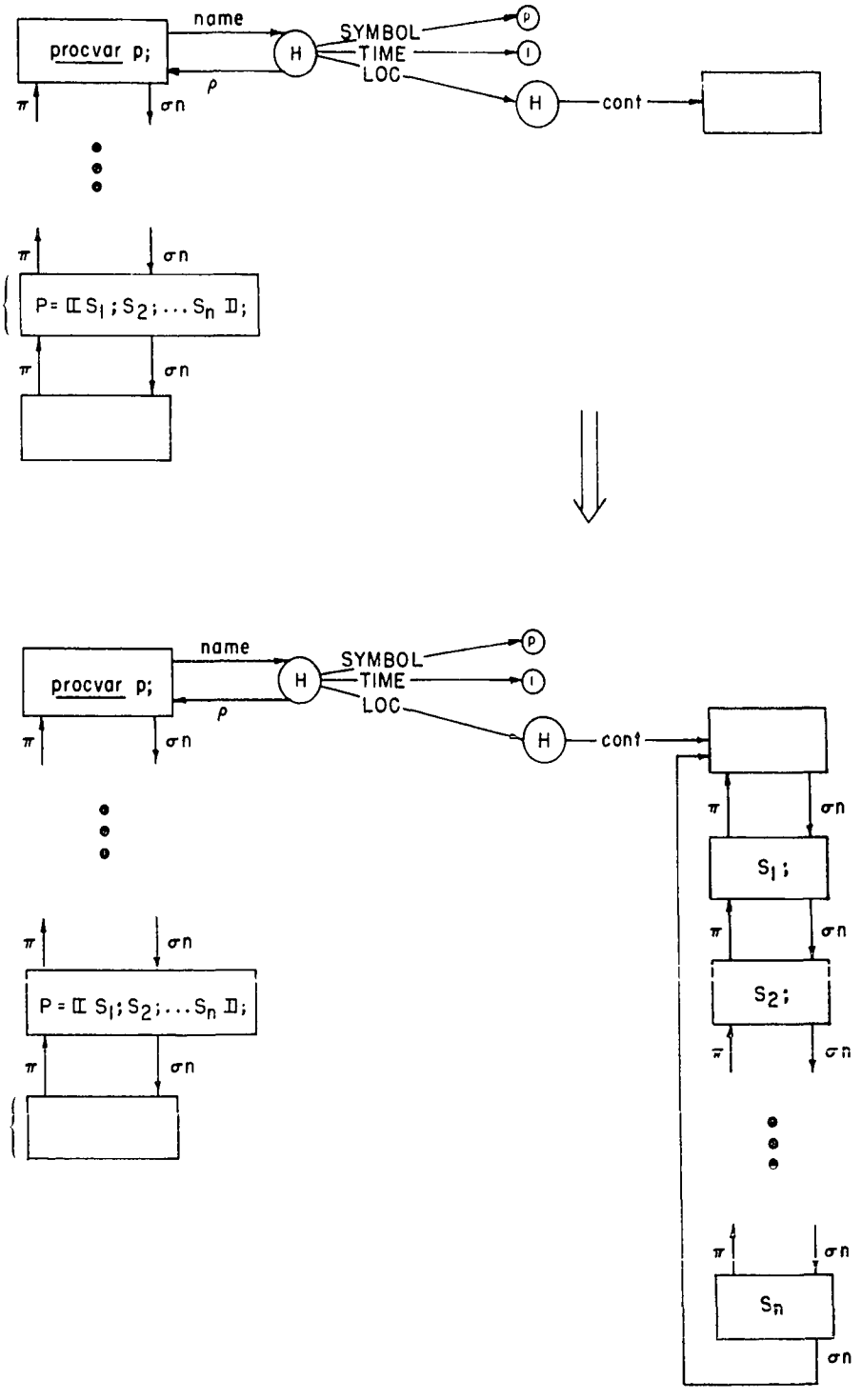


Diagram 2. Translation of a procedure text

property "identifier1" and Max value of the 'TIME' property has acquired an AGM representation of the translated text.

Case III.

State prior to execution = $(G_i, \sigma n^j)$, where the contents of σn^j = "identifier1 = [identifier2, close {clist}];".

The exact NAL code for this construct depends on the clist. Informally, this NAL code works as follows:

1. In all cases, the value of the 'cont' property of the subname of identifier2 is copied, via the COPY operation, to the 'cont' property of the subname of identifier1.
2. Closures are then modeled by the creation of names on the blank program header nodes. The two forms of closures are illustrated on the following page.

Call node

State prior to execution = (G_i, Ψ),

where the contents of Ψ = "call identifier;".

NAL code:

```
attachprop (Ψ, {('σB', COPY(propval (propval (propx (propset (N,
           'SYMBOL', identifier), 'TIME', Max), 'loc'),
           'cont')))});
attachprop (ΨσB, {('π', Ψ)});
attachprop (propx (N, 'SYMBOL', $C), {('val', propval (propx (N,
           'SYMBOL', $C), 'val')+1)});
```

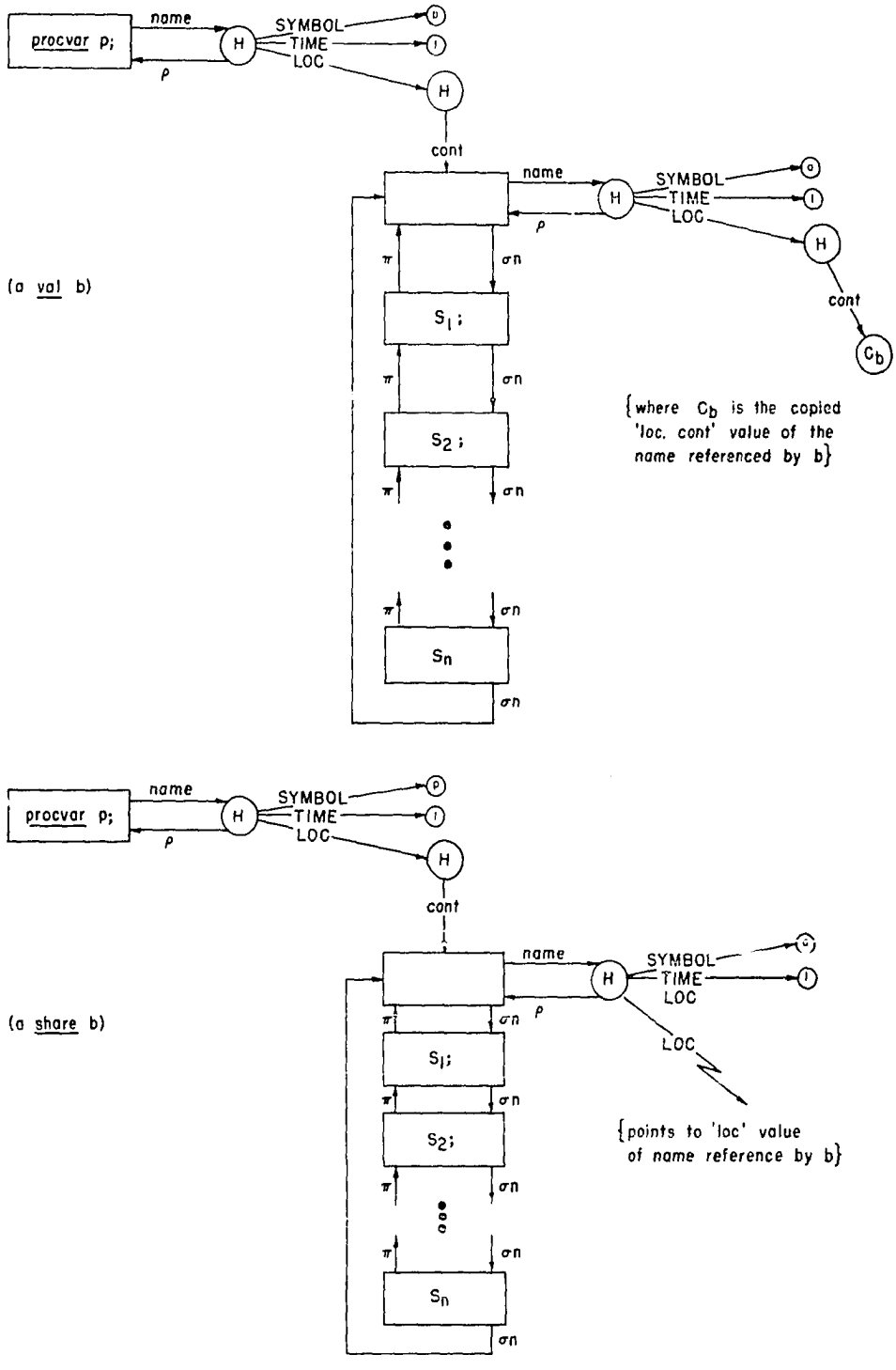


Diagram 3. Share and value closures

New state = (G_{i+1}, $\Psi\sigma B$), where G_{i+1} differs from G_i in that Ψ has acquired a σB property whose value is a copy of the 'cont' property of the subname of identifier1. In addition, $\Psi\sigma B$ acquires a π property whose value is Ψ and the 'val' property of the clock is also incremented by 1. The effect of a call is to make a copy of the code and then to resume execution in that copy.

Blank node

State prior to execution = (G_i, $\Psi\sigma B$),

where the contents of $\Psi\sigma B$ = "B".

The effect of executing a blank node is to replace the contents of the node with shading or B.

B ≡ 

B ≡ 

New state = (G_{i+1}, $\Psi\sigma B\sigma n$).

Shaded node

State prior to execution = (G_i, $\Psi\sigma B\sigma n^j$),

where the contents of $\Psi\sigma B\sigma n^j$ = "B".

NAL code:

destroy ($\{\Psi\sigma B\sigma n^j\}$);

New state = (G_{i+1}, $\Psi\sigma n$), where G_{i+1} differs from G_i in that ($\Psi\sigma B\sigma n^j$) and all nodes accessible from it (and inaccessible from Ψ with normal (σn) flow) are destroyed.

End node

State prior to execution = $(G_i, \sigma n^j)$,

where the contents of σn^j = "end".

NAL code:

destroy (propset (N, 'TIME', 1));

New state = $(G_{i+1}, \sigma n^{j+1})$, where G_{i+1} differs from G_i in that all names declared in the main program are destroyed.

DC node

State prior to execution = $(G_i, \sigma n^j)$,

where the contents of σn^j = "DC".

NAL code:

destroy (propset (N, 'SYMBOL', $C));

New state = $(G_{i+1}, \sigma n^{j+1})$, where G_{i+1} differs from G_i in that the system clock is destroyed.

Example 8

ML-4 Program

```
new a;
new b;
procvar p;
procvar q;
p = [[new a; a ← b; c ← 3]];
q = [p, close {(b val 3), (c share a)}];
a ← 1;
b ← 2;
call q
end
```

The partial execution trace for this program is given in Figures 12a-12d. The NAL code used to effect the state transformations is given as follows.

EXECUTE (G_0, σn) =

attachprop (σn, 'name', createname ({('SYMBOL', $C), ('val', 1)}));

attachprop (propx (N, 'SYMBOL', $C), 'ρ', σn);

EXECUTE (G_1, σn^2) =

attachprop (σn^2, 'name', createname ({('SYMBOL', a), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', a), 'TIME', Max), {('ρ', σn^2), ('loc', createname ({('cont', Ω)}))});

EXECUTE (G_2, σn^3) =

attachprop (σn^3, 'name', createname ({('SYMBOL', b), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', b), 'TIME', Max), {('ρ', σn^3), ('loc', createname ({('cont', Ω)}))});

EXECUTE (G_3, σn^4) =

attachprop (σn^4, 'name', createname ({('SYMBOL', p), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

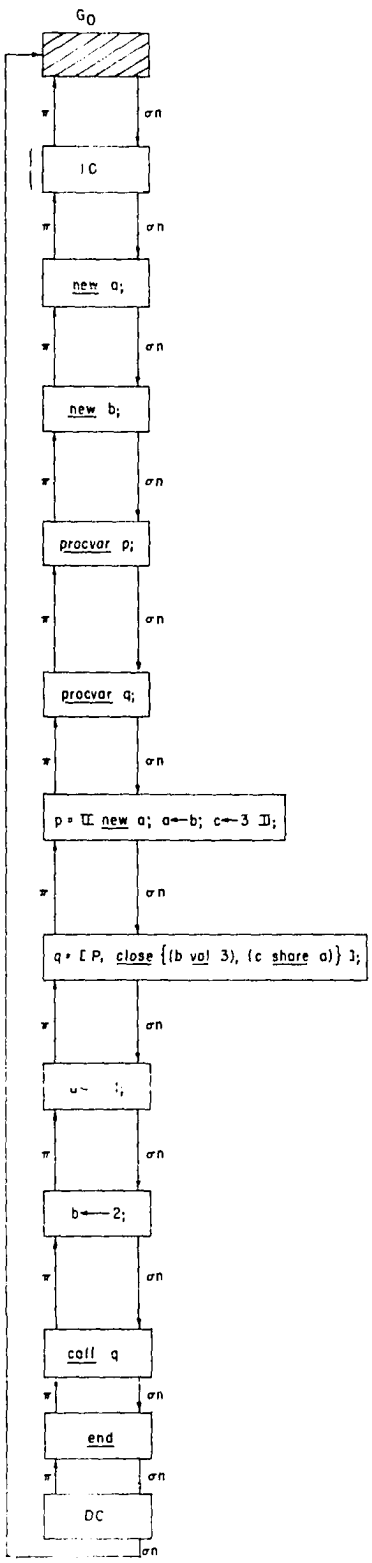


Figure 12a. Initial AGM representation for example 8 with state = (G_0, σ)

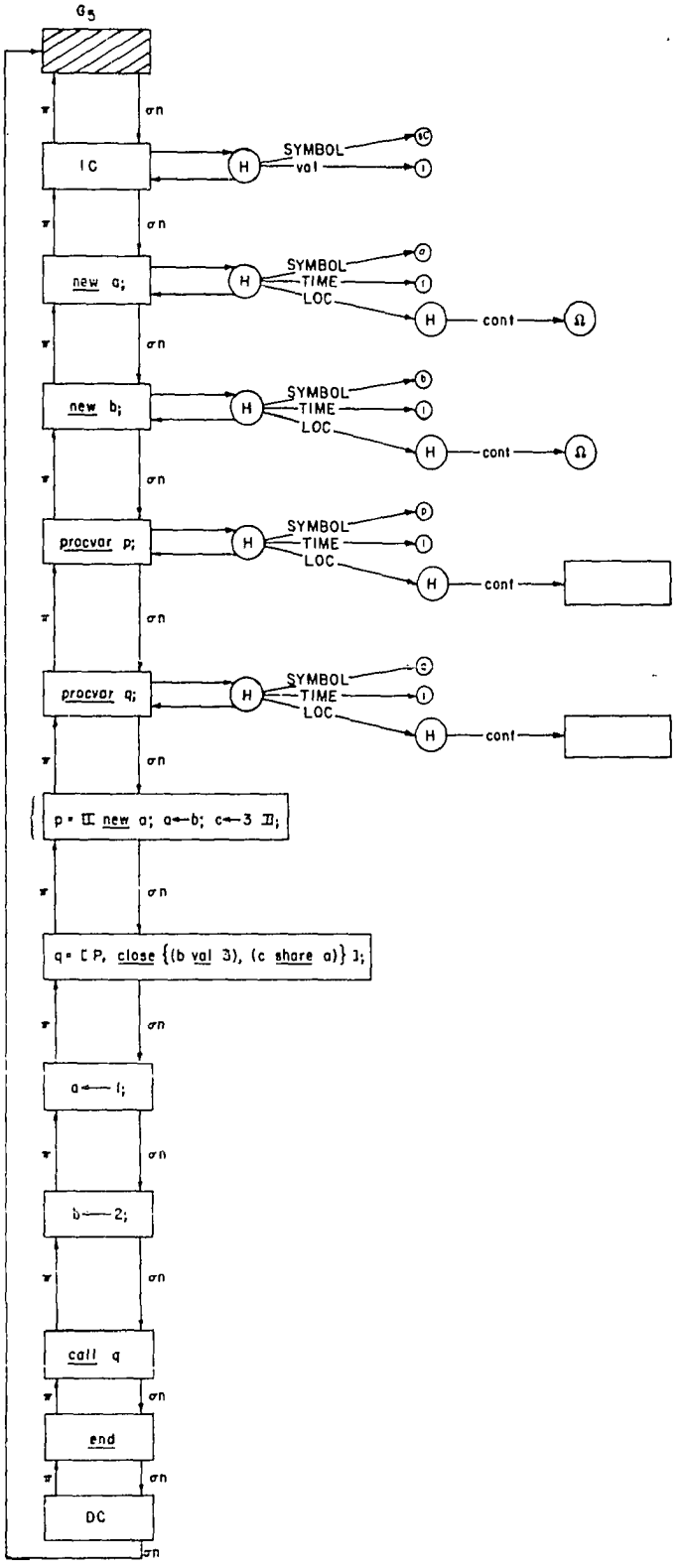


Figure 12b. Execution trace for example 8 with state = $(G_5, \sigma n^6)$

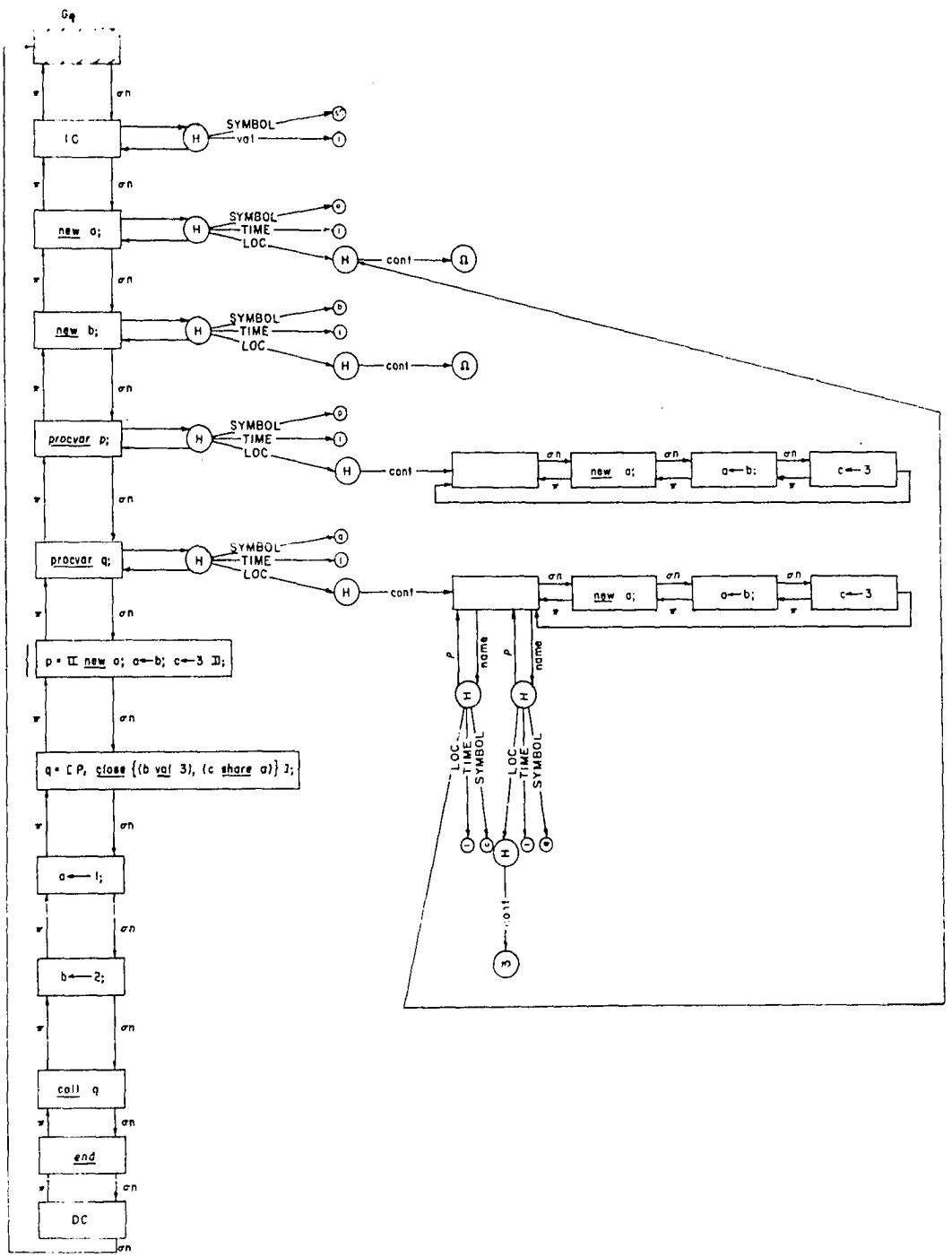


Figure 12c. Execution trace for example 8 with state = $(G_9, \sigma n^{10})$

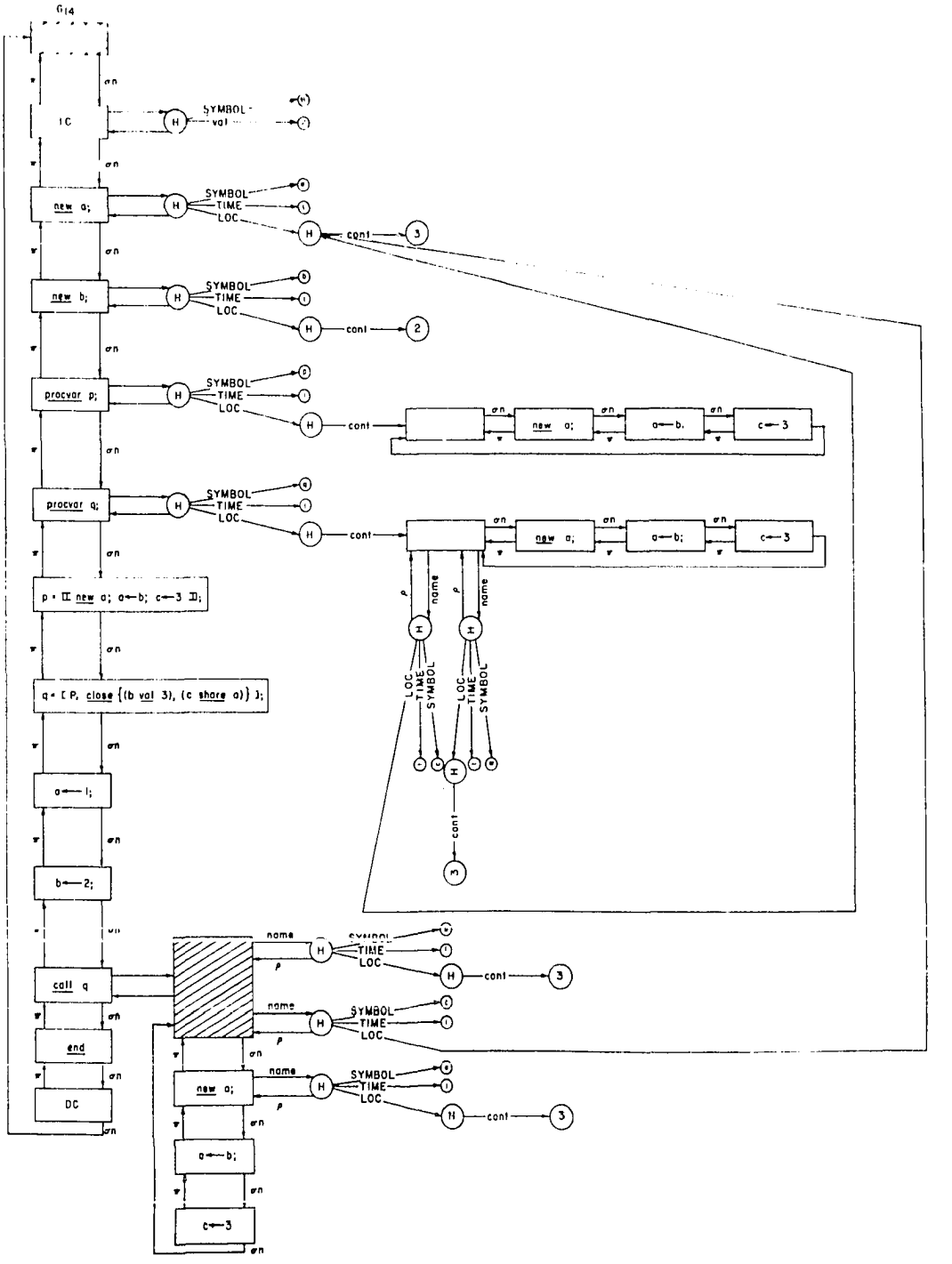


Figure 12d. Execution trace for example 8 with state = $(G_{14}, \sigma n^{10} \sigma B \sigma n^{4})$

attachprop (propx (propset (N, 'SYMBOL', p), 'TIME',
Max), {('ρ', σn^4), ('loc', createname ({('cont',
B)})) });

EXECUTE (G_4, σn^5) =

attachprop (σn^5, 'name', createname ({('SYMBOL', q), ('TIME',
propval (propx (N, 'SYMBOL', $C), 'val'))}));
attachprop (propx (propset (N, 'SYMBOL', q), 'TIME',
Max), {('ρ', σn^5), ('loc', createname ({('cont',
B)})) });

EXECUTE (G_5, σn^6) =

attachprop (propval (propval (propx (propset (N, 'SYMBOL', p),
'TIME', Max), 'loc'), 'cont'), {('σn', H(X))});
attachprop (H(X), {('π', propval (propval (propx (propset (N,
'SYMBOL', p), 'TIME', Max), 'loc'), 'cont'))});
attachprop (T(X), {('σn', propval (propval (propx (propset (N,
'SYMBOL', p), 'TIME', Max), 'loc'), 'cont')))});
(where X = TRANSLATE [[new a; a ← b; c ← 3]])

EXECUTE (G_6, σn^7) =

attachprop (propval (propx (propset (N, 'SYMBOL', q), 'TIME', Max),
'loc'), {('cont', COPY(propval (propval (propx (propset (N,
'SYMBOL', p), 'TIME', Max), 'loc'), 'cont')))});

attachprop (propval (propval (propx (propset (N, 'SYMBOL', q), 'TIME', Max), 'loc'), 'cont'), 'name', createname ({('SYMBOL', b), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val')), ('ρ', propval (propval (propx (propset (N, 'SYMBOL', q), 'TIME', Max), 'loc'), 'cont')), ('loc, createname ({('cont', 3)}))}));

attachprop (propval (propval (propx (propset (N, 'SYMBOL', q), 'TIME', Max), 'loc'), 'cont'), 'name', createname ({('SYMBOL', c), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val')), ('ρ', propval (propval, (propx (propset (N, 'SYMBOL', q), 'TIME', Max) 'loc'), 'cont')), ('loc', propval (propx (propset (N, 'SYMBOL', a), 'TIME', Max), 'loc'))}));

EXECUTE (G_7, σn^8) =

attachprop (propval (propx (propset (N, 'SYMBOL', a), 'TIME', Max), 'loc'), {('cont', 1)});

EXECUTE (G_8, σn^9) =

attachprop (propval (propx (propset (N, 'SYMBOL', b), 'TIME', Max), 'loc'), {('cont', 2)});

EXECUTE (G_9, σn^{10}) =

attachprop (σn^{10}, {('σB', COPY(propval (propval (propx (propset (N, 'SYMBOL', q), 'TIME', Max), 'loc'), 'cont')))});

attachprop ($\sigma n^{10}\sigma B$, {('π', σn^{10})});

attachprop (propx (N, 'SYMBOL', $C), {('val', propval (propx (N, 'SYMBOL', $C), 'val')+1)});

EXECUTE (G_{10}, $\sigma n^{10}\sigma B$) =

(shade header node)

EXECUTE (G_{11}, σn^{10}, $\sigma B\sigma n$) =

attachprop ($\sigma n^{9}\sigma B\sigma n$, 'name', createname ({('SYMBOL', a), ('TIME', propval (propx (N, 'SYMBOL', $C), 'val'))}));

attachprop (propx (propset (N, 'SYMBOL', a), 'TIME', Max), {('p', $\sigma n^{9}\sigma B\sigma n$), ('loc', createname ({('cont', Ω)}))});

EXECUTE (G_{12}, σn^{10}, $\sigma B\sigma n^{2}$) =

attachprop (propval (propx (propset (N, 'SYMBOL', a), 'TIME', Max), 'loc'), {('cont', propval (propval (propx (propset (N, 'SYMBOL', b), 'TIME', Max), 'loc'), 'cont'))});

EXECUTE (G_{13}, σn^{10}, $\sigma B\sigma n^{3}$) =

attachprop (propval (propx (propset (N, 'SYMBOL', c), 'TIME', Max), 'loc'), {('cont', 3)});

EXECUTE (G_{14}, σn^{10}, $\sigma B\sigma n^{4}$) =

destroy ({$\sigma n^{10}\sigma B\sigma n^{4}$});

EXECUTE (G_{15}, σn^{11}) =

destroy (propset (N, 'TIME', 1));

EXECUTE (G_{16}, σn^{12}) =

```
destroy (propset (N, 'SYMBOL', $C));
```

Example 9

ML-4 Program

```
new b;
new c;
new d;
new a;
procvar p;
procvar q;
procvar r;
p = [[c ← a; call q]];
q = [[d ← a]];
r = [p, {(a share b)}];
a ← 1;
b ← 2;
call r
end
```

Example 9 is designed to demonstrate the effect of closures on subsequently called procedures. In this example, procedure r is called from the main program. This procedure has three nonlocals, c, a, and q. The nonlocal c is bound by default to the c declared in the main program. The a is explicitly bound by a share closure with b. The procedure q which is called from within r is bound by default to the procedure q declared in the main program. Procedure q itself has two nonlocals, d and a, both bound by default. The d is bound to the d declared in the main program, but since q is called from within r, the a is bound to the a closed in r. A snapshot of the AGM state at $P = \sigma n^{11} \sigma B \sigma n^2 \sigma B \sigma n^2$ is given in Figure 13.

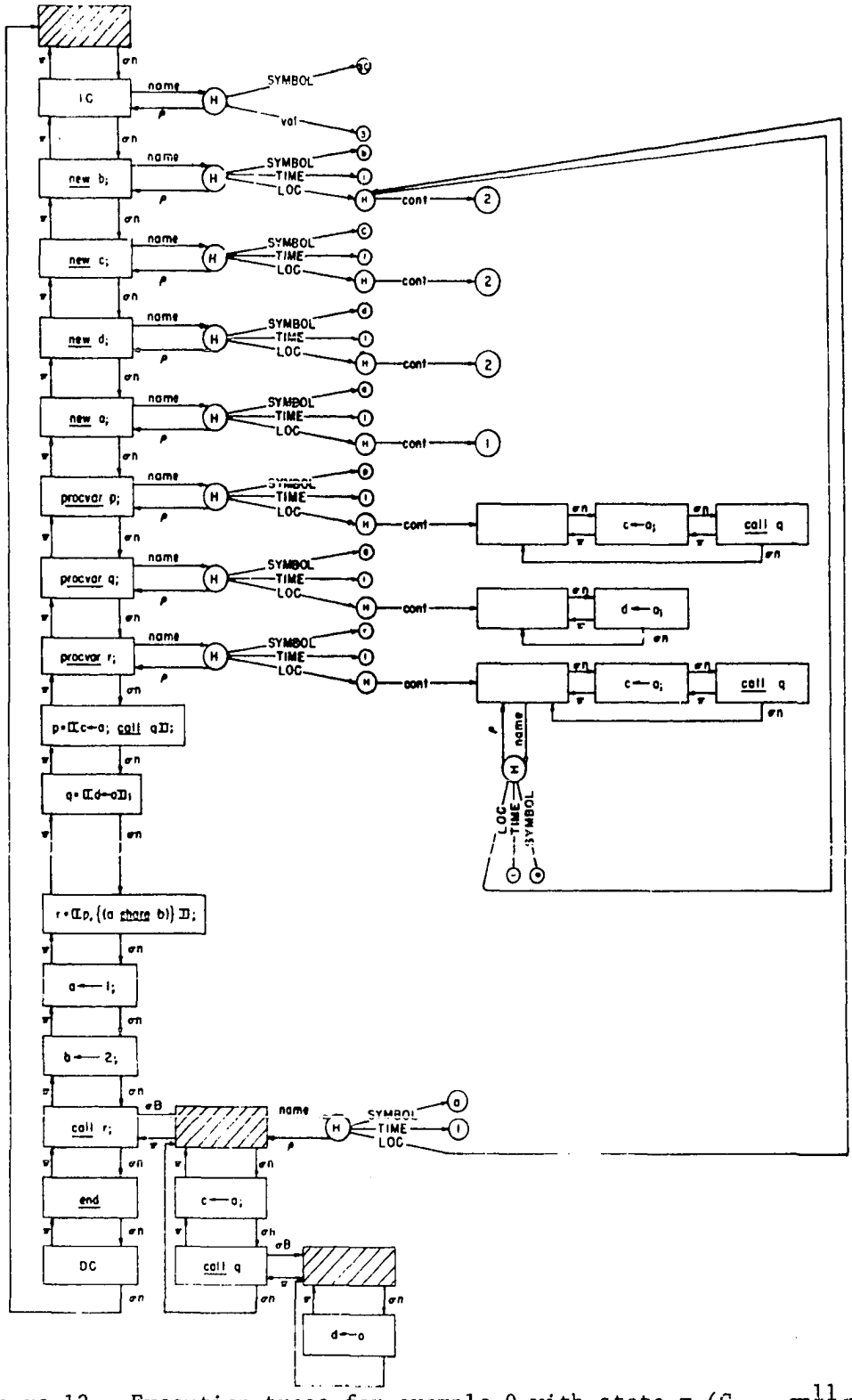


Figure 13. Execution trace for example 9 with state = $(G_{16}, \sigma n^{11}\sigma B\sigma n^{2}\sigma B\sigma n^{2})$

CHAPTER V. NAME ACCESSING IN THE SYMBOL-2R PROGRAMMING LANGUAGE

The purpose of this chapter is to apply the AGM modeling techniques to an actual programming language. The author has chosen the SYMBOL-2R Programming Language (SPL) as a vehicle of study for this objective. Since SPL offers a rich variety of name accessing capabilities, many of which are present in more common high-level languages, it seems well suited for its role as a case study.

The SYMBOL-2R Computing System was designed and constructed by Fairchild's Semiconductor Digital Systems Research Group beginning about 1964. It was purchased by Iowa State University in 1970 through a grant from the National Science Foundation. Since then, SYMBOL-2R has been the focal point of a research effort whose goals include the evaluation of the design features incorporated in SYMBOL-2R. SPL itself played a rather large role in determining some of these design features since SYMBOL-2R represents one of the earlier attempts at language-directed computer design. For a detailed discussion of the SYMBOL-2R system, the reader is referred to (1, 10, 32, 57, 58).

For the purposes of this chapter, we will consider the SYMBOL-2R Language Reference Manual (56) as the defining document for SPL. In an attempt to clarify some of the issues discussed, we will take some liberties with the syntax of SPL, but will adhere to the semantic description as specified in the Reference Manual. In the pure SPL form it is assumed that identifiers are declared by their use. In the modified

SPL form used in this chapter it is required that identifiers be explicitly declared via the "new identifier;" construct. This explicit declaration of identifiers has no effect on the semantic intent of name accessing in SPL. An example of this syntactic modification is illustrated in Figure 14. The symbol "≠" is used to denote the end of an SPL program.

```
a ← 1;          new a;
b ← a;          new b;
≠               a ← 1;
                b ← a
                ≠
```

(a) Pure SPL (b) Modified SPL

Figure 14. Explicit declaration of identifiers

SPL is a general purpose, high-level language which supports several very common language concepts. These include:

1) block structure,
2) procedures,
3) parameters with "call by name" resolution,
4) function type procedures,
5) transfers and conditionals,
6) structured data,
7) generalized input-output, and
8) assignment.

In addition, SPL supports some less common language features such as

1) nondefault scope rules and

2) indirect references via "links".

In the present discussion of SPL, emphasis will be placed on those language features which contribute directly to the name accessing capabilities of the language. In particular, we will concentrate on the following:

1) block structure with nondefault scope rules,

2) procedures and parameters, including function-type procedures, and

3) indirect references created via "links".

One of the more interesting and crucial aspects of name accessing concerns the time at which information is bound to a name. This is generally referred to as the binding time of information. In terms of the AGM, information is represented by (P,V) pairs so we will refer to the binding of a property to a name or the binding of a particular value to a property. In general terms, we can distinguish three broad classes of binding times: translation or compilation time, load time and execution time. In SPL, translation and load time are indistinguishable and will simply be referred to as translation time. Execution time binding can be further subdivided into finer classifications such as block entry time, procedure call time, reference time, etc.

In specifying the semantics of ML-1 - ML-4, we chose to delay, until execution time, the binding of a name to its creation node and the binding of a subname (location) to a name. This is not the case in SPL where names and locations are assigned at translation time. This

implies that the binding of the 'name', 'ρ', 'loc', 'cont', and 'SYMBOL' properties to their associated values are accomplished via the TRANSLATE function rather than the EXECUTE function. In addition, since SPL is a block structured language which permits the redeclaration of a name with similar 'SYMBOL' property values, a system clock must be maintained at translation time to calculate 'TIME' values for names. The system clock for SPL must be a nondecrementing clock due to the fact that SPL supports retention of names outside of their scope. The necessity of a nondecrementing clock is illustrated in the diagram in Figure 15. The boxes in this diagram indicate blocks, with BLOCK 2 and BLOCK 3 nested within BLOCK 1. If the names accessible in BLOCK 2 are retained, even when executing in BLOCK 3, names with similar 'SYMBOL' properties must be marked with different 'TIME' properties to avoid clashes.

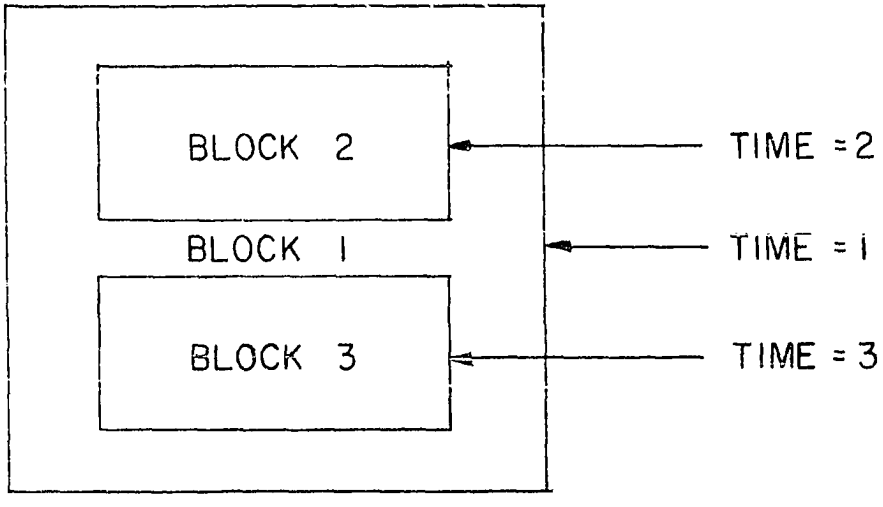


Figure 15. Retention and the nondecrementing clock

In order to take advantage of the extra overhead incurred by the translation of SPL programs into AGM form, program nodes which contain reference occurrences of names are given a 'TIME' property, whose value is the value of the system clock for that block. This 'TIME' property is used to resolve references during program execution.

Example 10

SPL Program

```
new a;
new b;
a ← 1;
b ← a;
a ← 2
≠
```

A partial execution trace for the program listed in Example 10 is given in Figures 16a-16e.

The program nodes containing "new a;" and "new b;" have a null effect on the accessing graph when they are executed. In executing a node of this type the state transformation is then $(G_i, \sigma^j) \rightarrow (G_i, \sigma^{j+1})$. The 'name' property associated with a node of this type is acquired by N or added to the π chain via execution.

Names in SPL have two intrinsic properties, their 'SYMBOL' and 'TIME' properties. In addition, the modified SPL form used in this chapter requires a declaration for each name that is used in a block. This stipulation results in a name accessing situation similar to the one discussed in ML-3. In particular, the resolution of a given identifier appearing in program node σn^j can be accomplished with

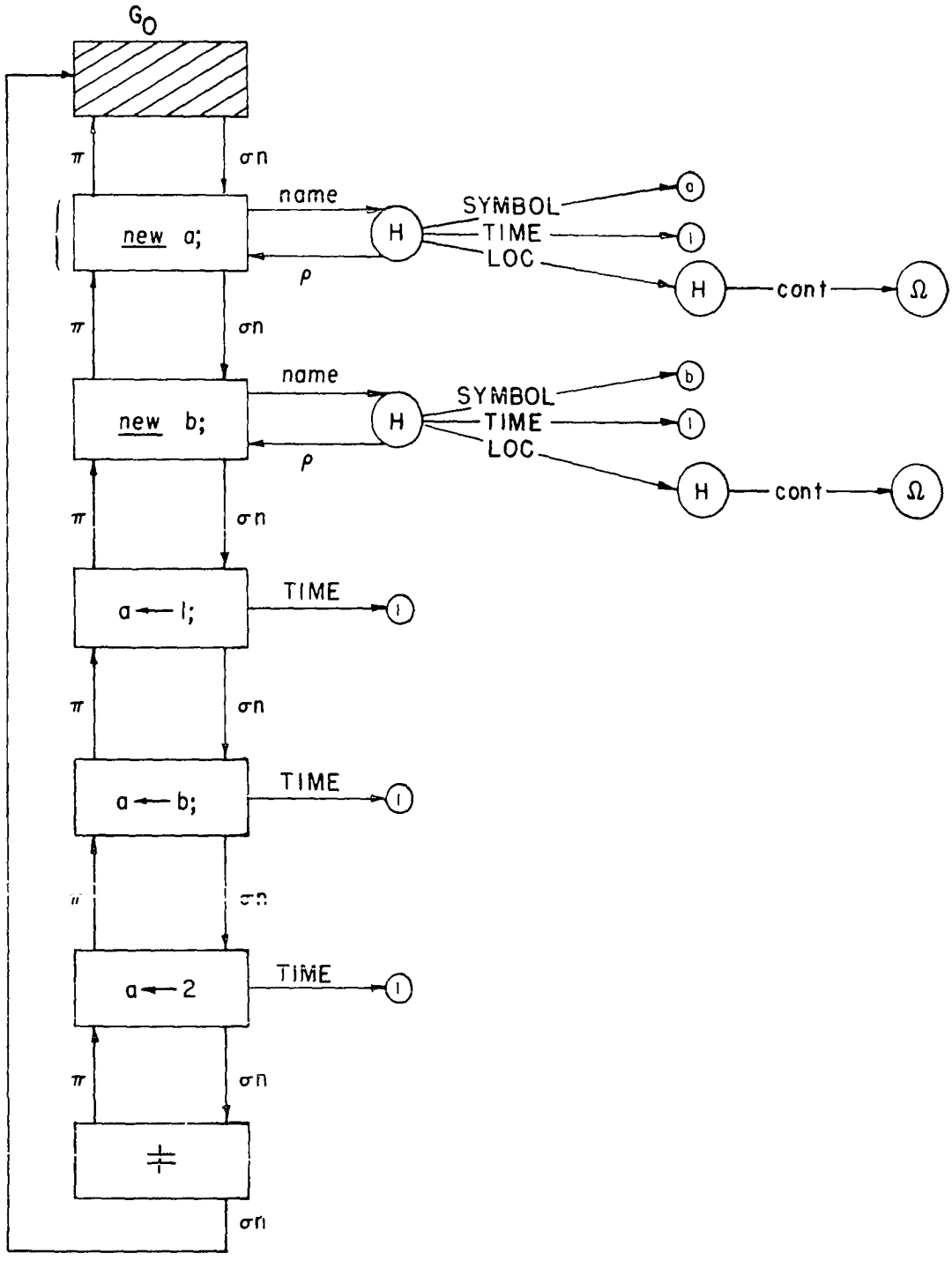


Figure 16a. Initial AGM representation for example 10 with state = (G_0, σn)

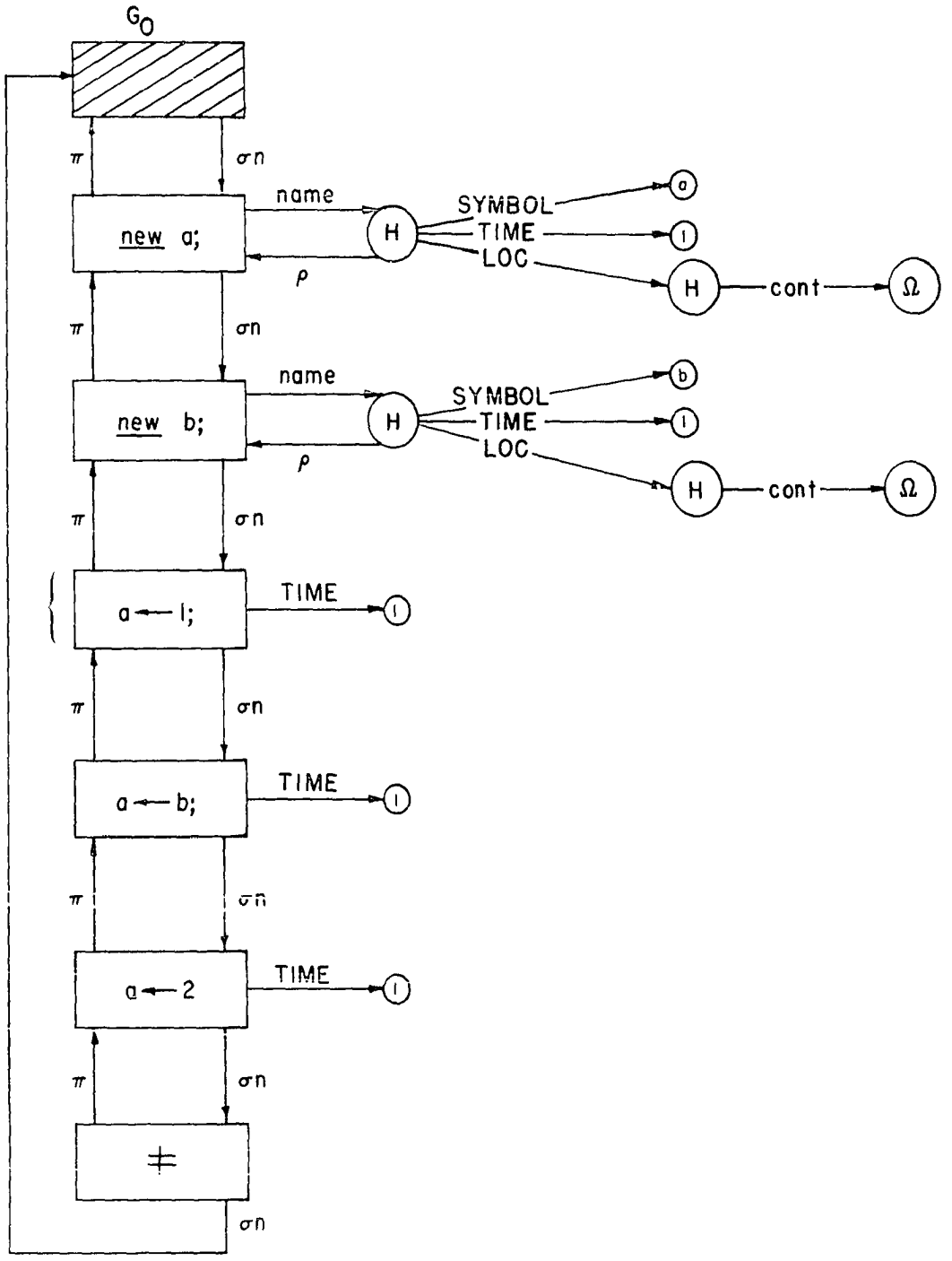


Figure 16b. Execution trace for example 10 with state = $(G_0, \sigma n^3)$

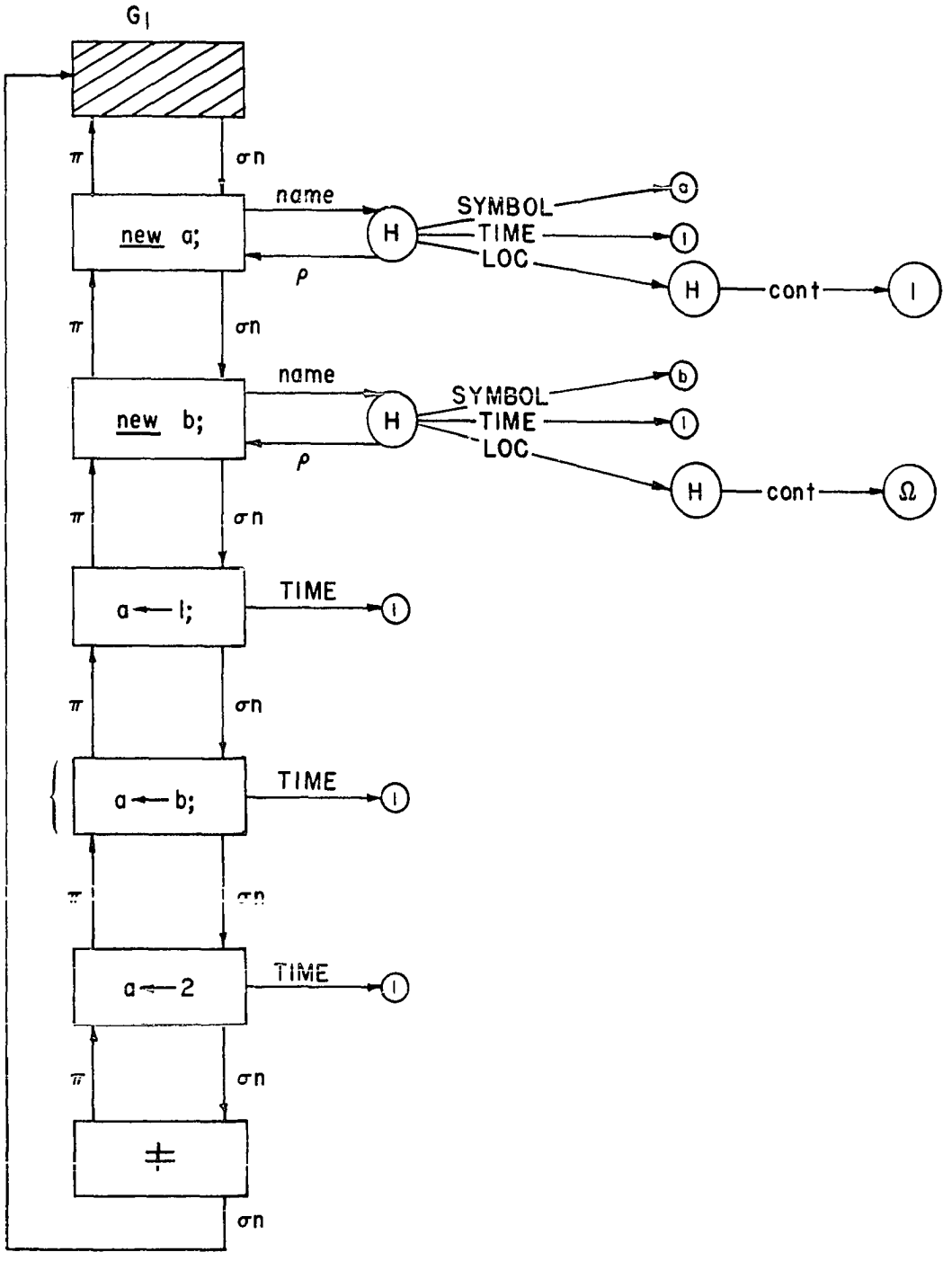


Figure 16c. Execution trace for example 10 with state = $(G_1, \sigma n^4)$

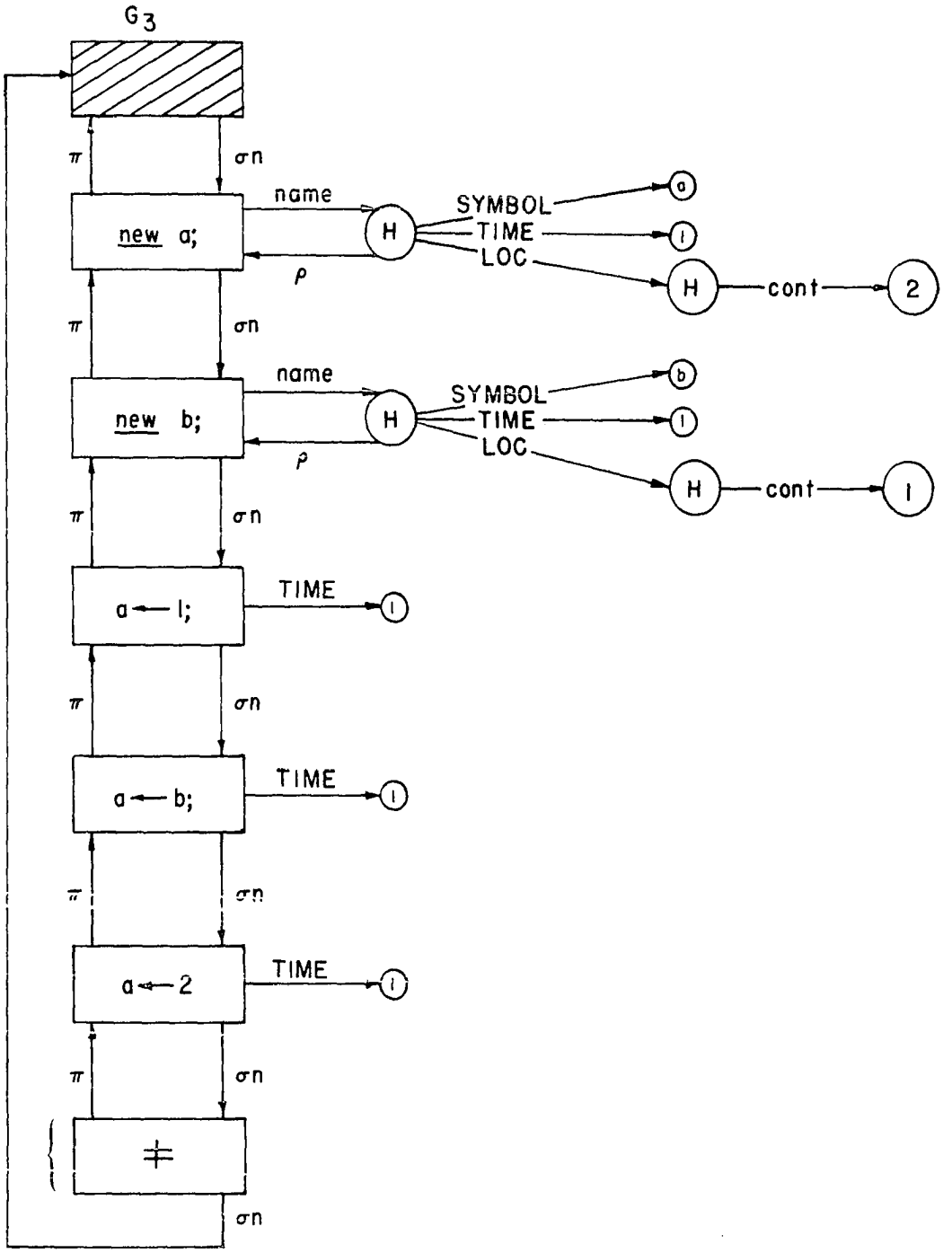


Figure 16d. Execution trace for ecample 10 with state = $(G_3, \sigma n^6)$

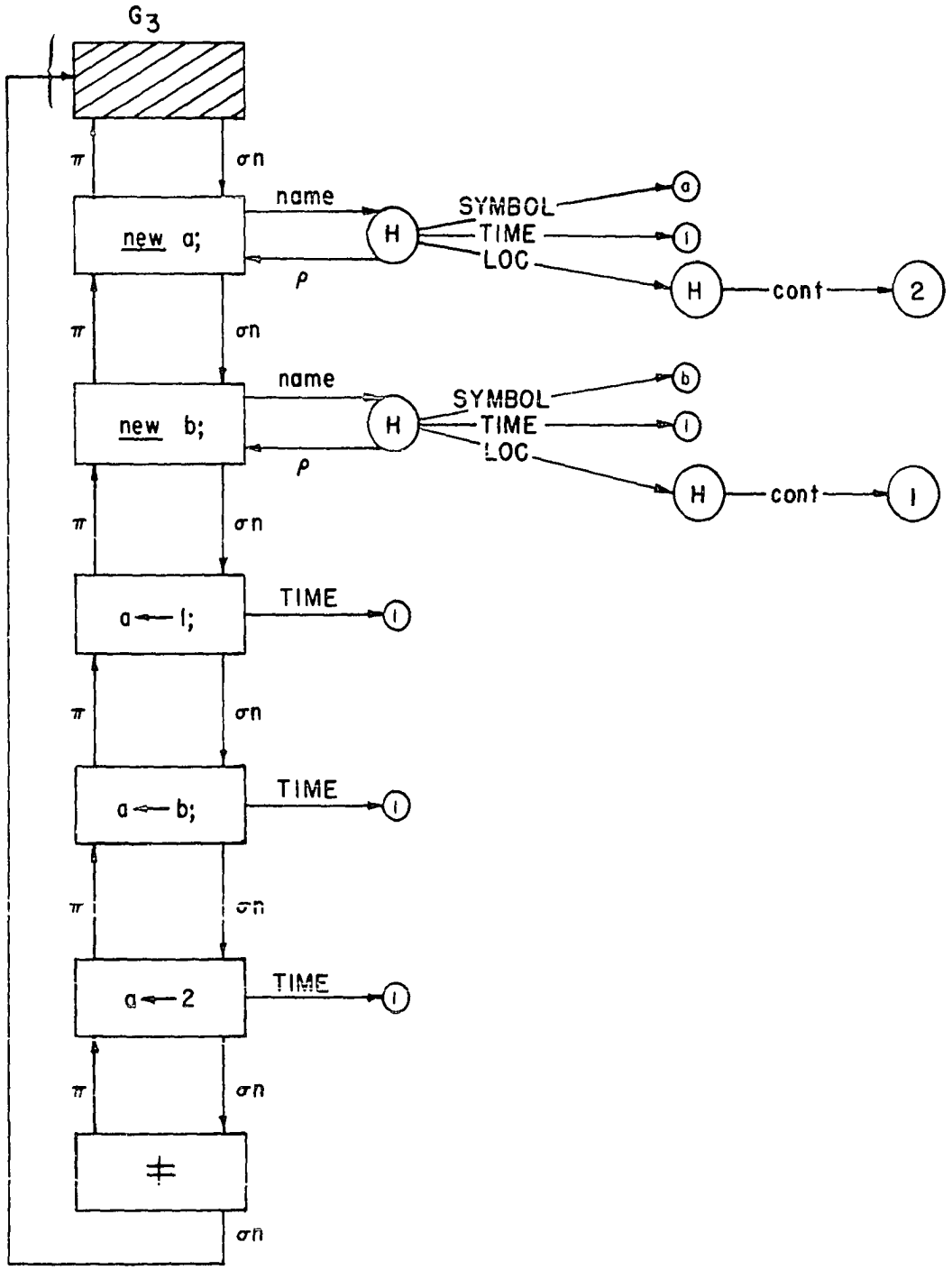


Figure 16e. Execution trace for example 10 with state = $(G_3, \sigma n^7)$

propx (propset (N, 'TIME', propval (σn^j, 'TIME')),
'SYMBOL', identifier)

or

propx (propset (N, 'SYMBOL', identifier), 'TIME',
propval (σn^j, 'TIME')).

For this treatment, the first alternative above will be used in all examples. Note that the 'TIME' property of the program node is used explicitly in the name accessing. Execution of Example 10 proceeds as follows.

EXECUTE (G_0, σn^3) =

attachprop (propval (propx (propset (N, 'TIME',
propval (σn^3, 'TIME')), 'SYMBOL', a), 'loc'), {('cont',
1)});

New state = (G_1, σn^4).

EXECUTE (G_1, σn^4) =

attachprop (propval (propx (propset (N, 'TIME', propval (σn^4,
'TIME')), 'SYMBOL', b), 'loc'), {('cont',
propval (propval (propx (propset (N, 'TIME',
propval (σn^4, 'TIME')), 'SYMBOL', a), 'loc'),
'cont'))});

New state = (G_2, σn^5).

EXECUTE (G_2, σn^5) =

attachprop (propval (propx (propset (N, 'TIME', propval (σn^5, 'TIME')), 'SYMBOL', a), 'loc'), {('cont', 2)});

New state = (G_3, σn^6).

Just as names are not created during execution, neither are they destroyed. Consistent with this, the execution of a "≠" program node has a null effect on the accessing graph. Note that even the 'cont' values are retained.

Example 11

SPL Program

```
new a;
new b;
a ← 1;
b ← 2;
block
    new b;
    global a;
    b ← a;
    a ← 3
end;
b ← a
≠
```

A partial execution trace for the program listed in Example 11 is given in Figures 17a-17d.

The global declaration in SPL is similar to the "near" construct introduced in ML-3 except that a global declaration may extend the scope of a variable only one level (or block) outward. Global's are therefore unable to create the type of scoping holes that the "near" construct permitted. The resolution of global's (i.e., global linking) is

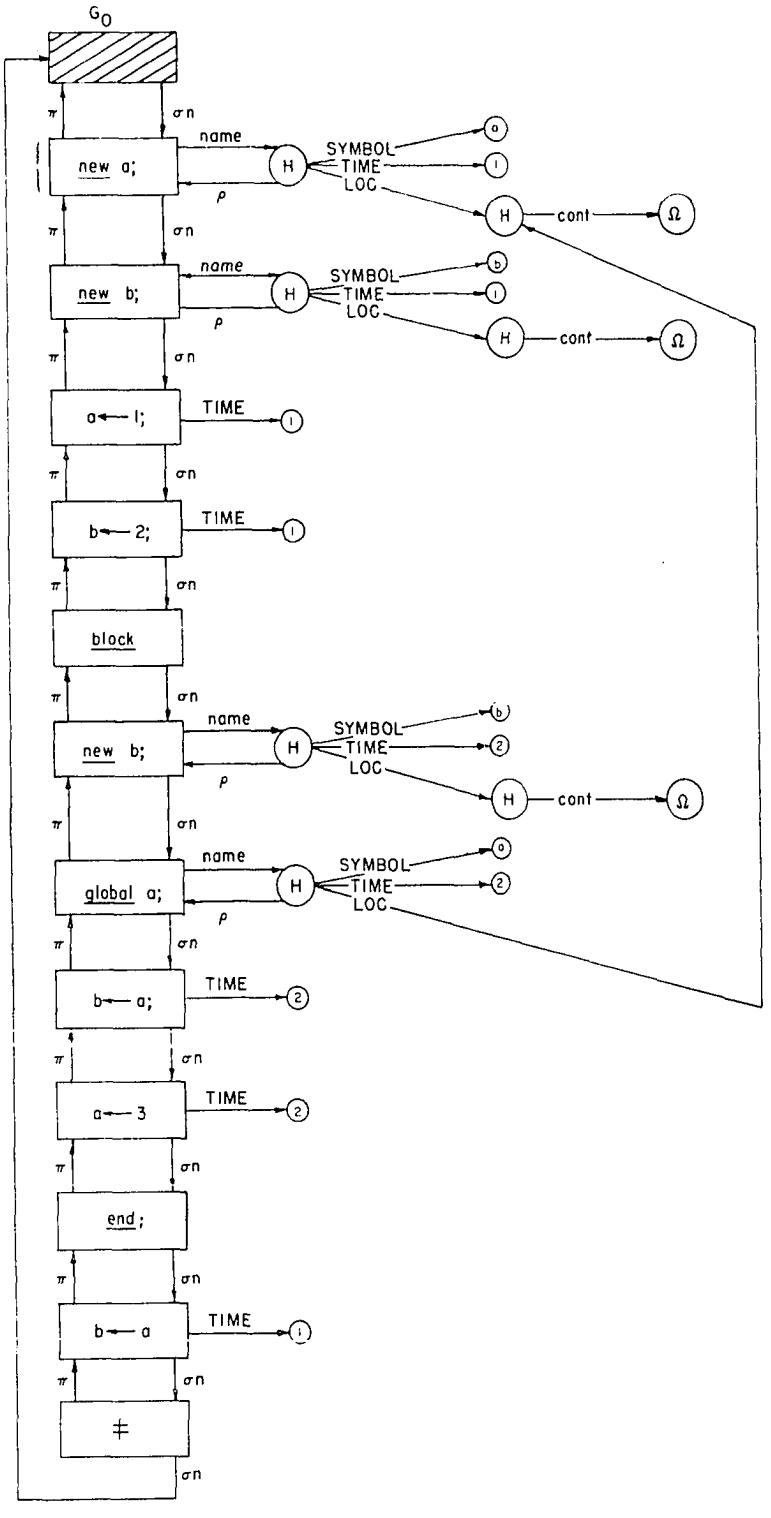


Figure 17a. Initial AGM representation for example 11 with state = (G_0, σn)

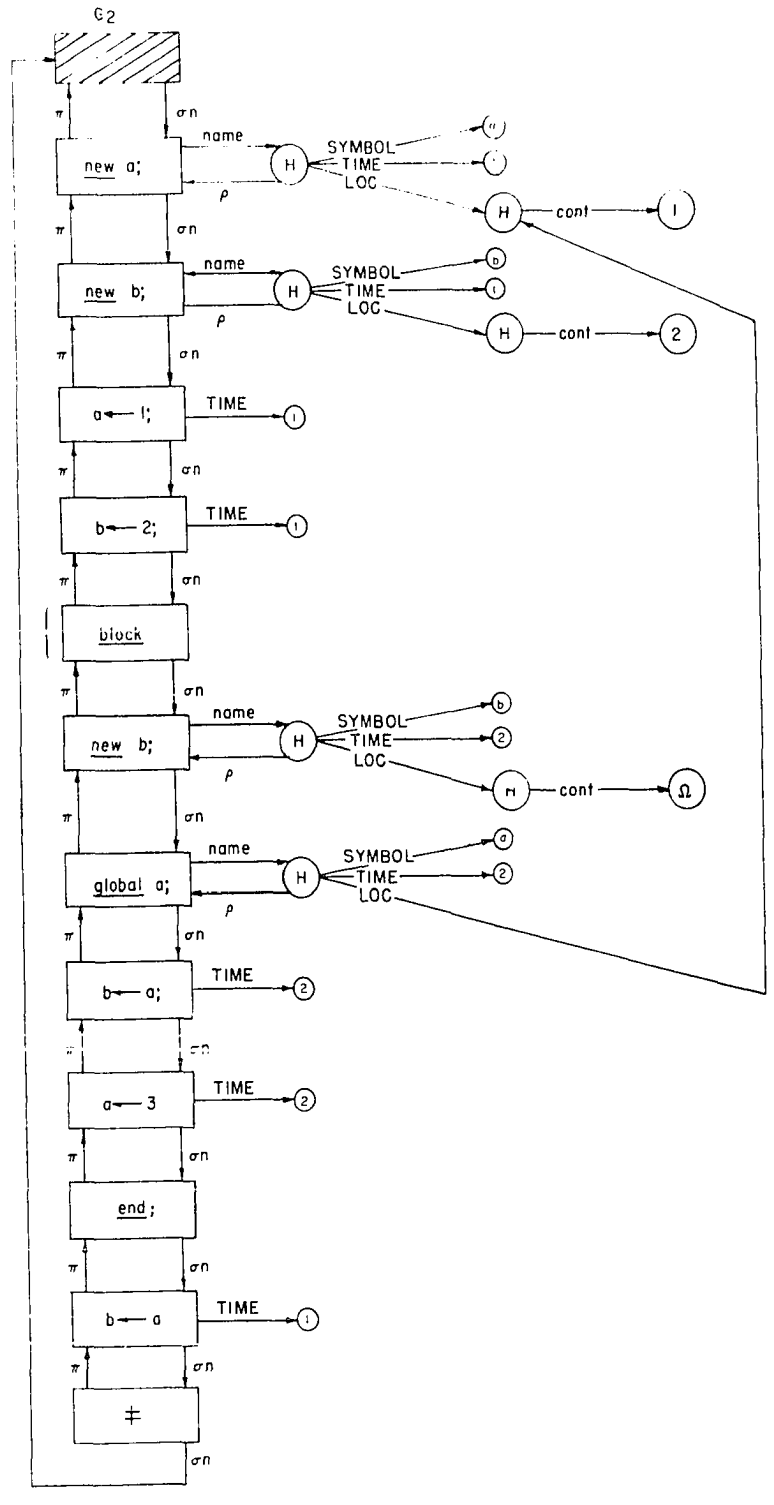


Figure 17b. Execution trace for example 11 with state = $(G_2, \sigma n^5)$

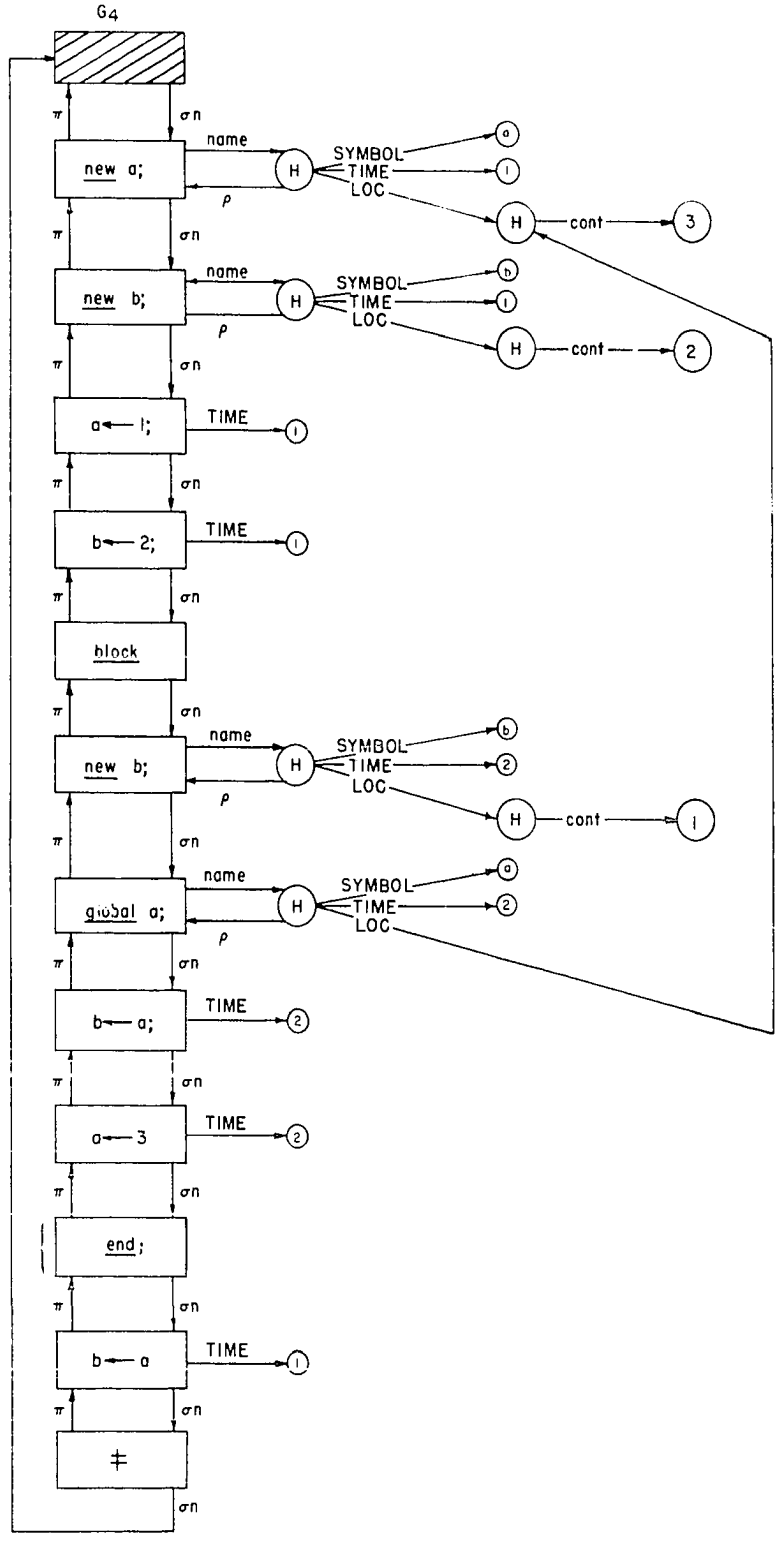


Figure 17c. Execution trace for example 11 with state = $(G_4, \sigma n^{10})$

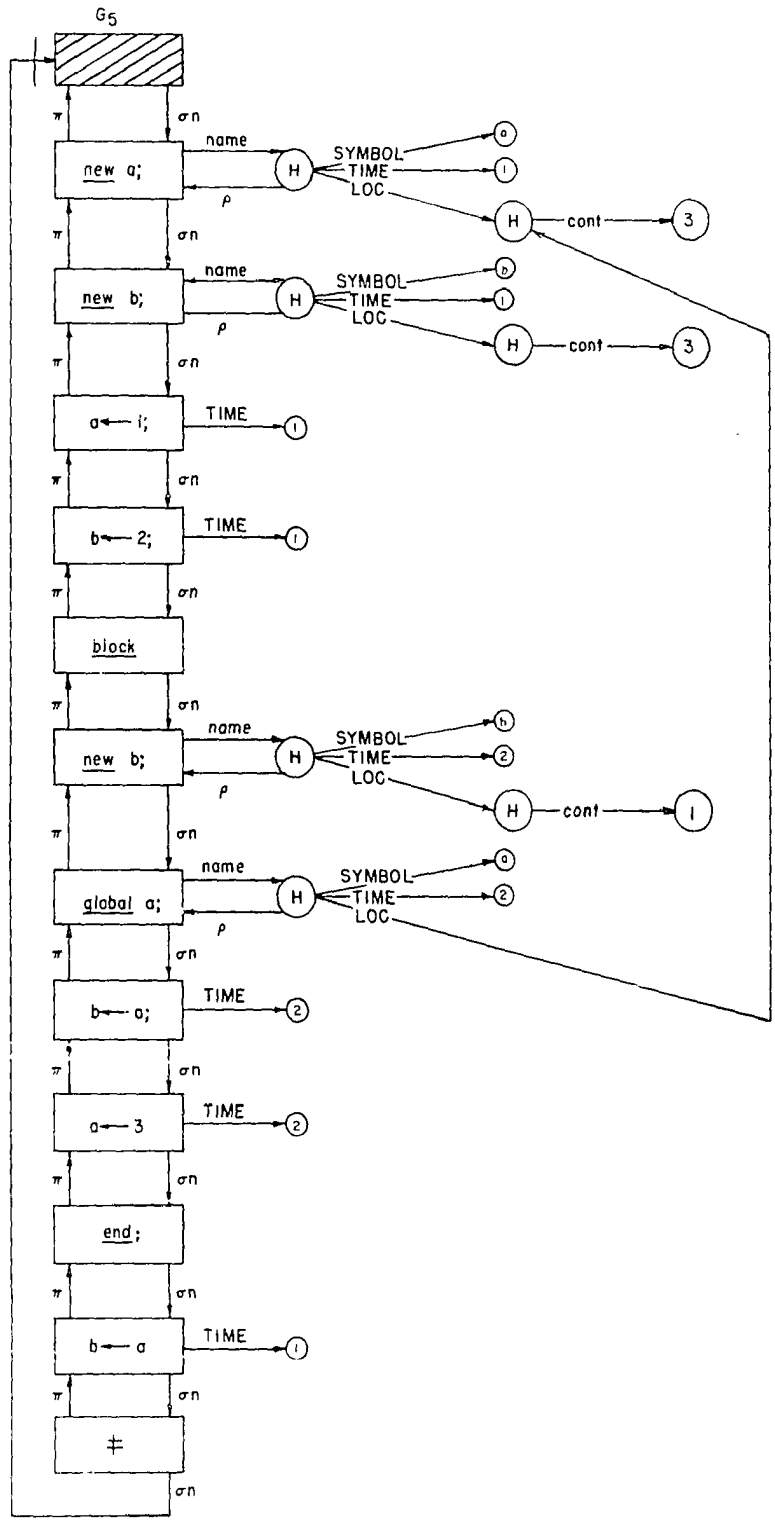


Figure 17d. Execution trace for example 11 with state = $(G_5, \sigma n^{13})$

performed at translation time in SPL.

In the partial execution trace for Example 11 note that identifier "a" of the inner block shares the location of the identifier "a" in the outer block.

The block nodes have no effect during execution. They are only instrumental in maintaining the system clock during translation. Since global linking is also performed during translation, global node names are simply acquired. The end nodes also have no effect since all names are retained in SPL.

Example 12

SPL Program

```
new a;
procedure p;
    global a;
    new b;
    b ← a
end;
a ← 1;
block
    global p;
    new a;
    a ← 2;
    call p
end
∓
```

A partial execution trace of the program listed in Example 12 is given in Figures 18a-18d.

The creation of names defined in SPL procedures is also a function of the translation routine. Procedural bodies are translated as encountered in the text and global variables are treated as they are in

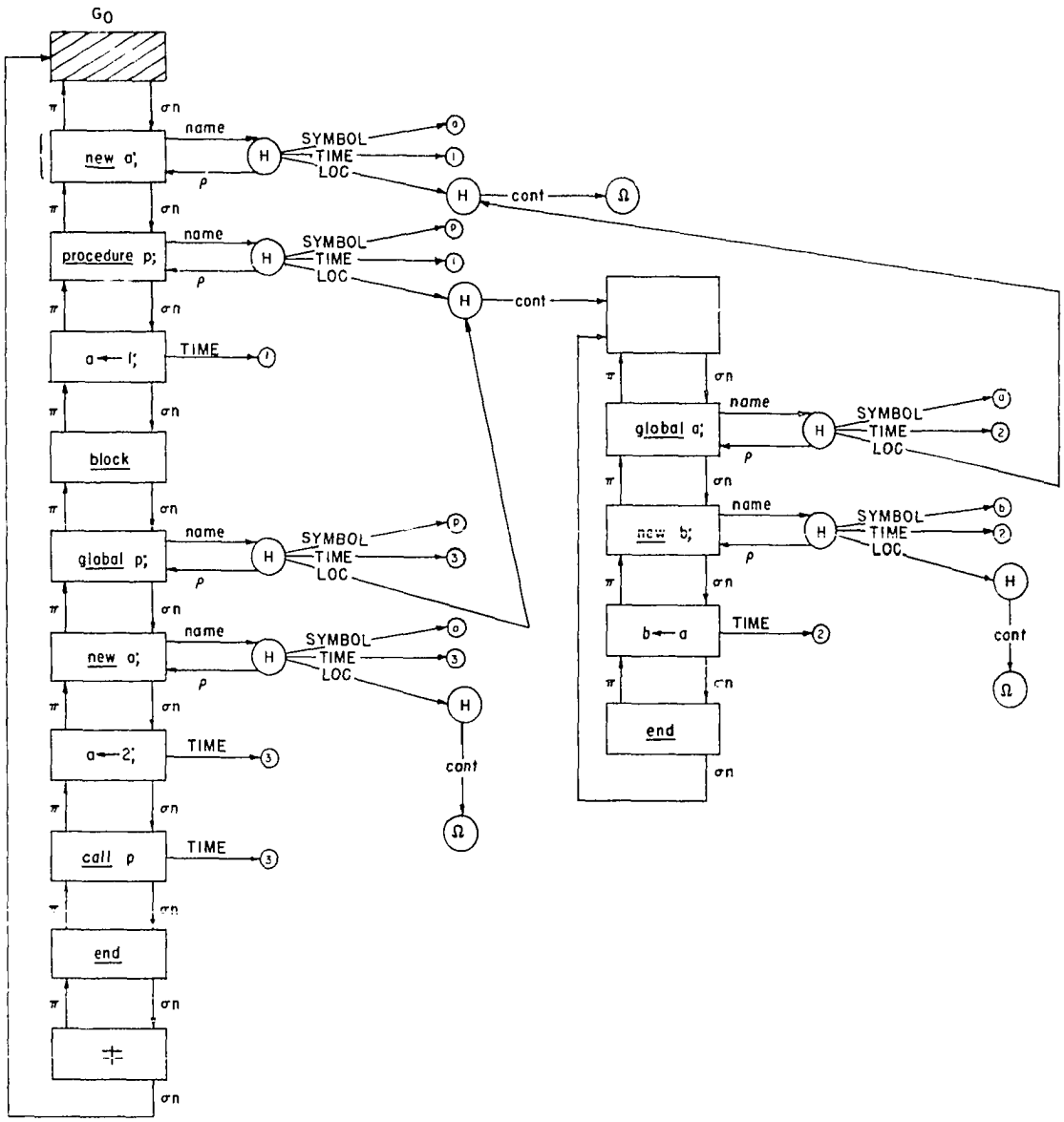


Figure 18a. Initial AGM representation for example 12 with state = (G_0, σn)

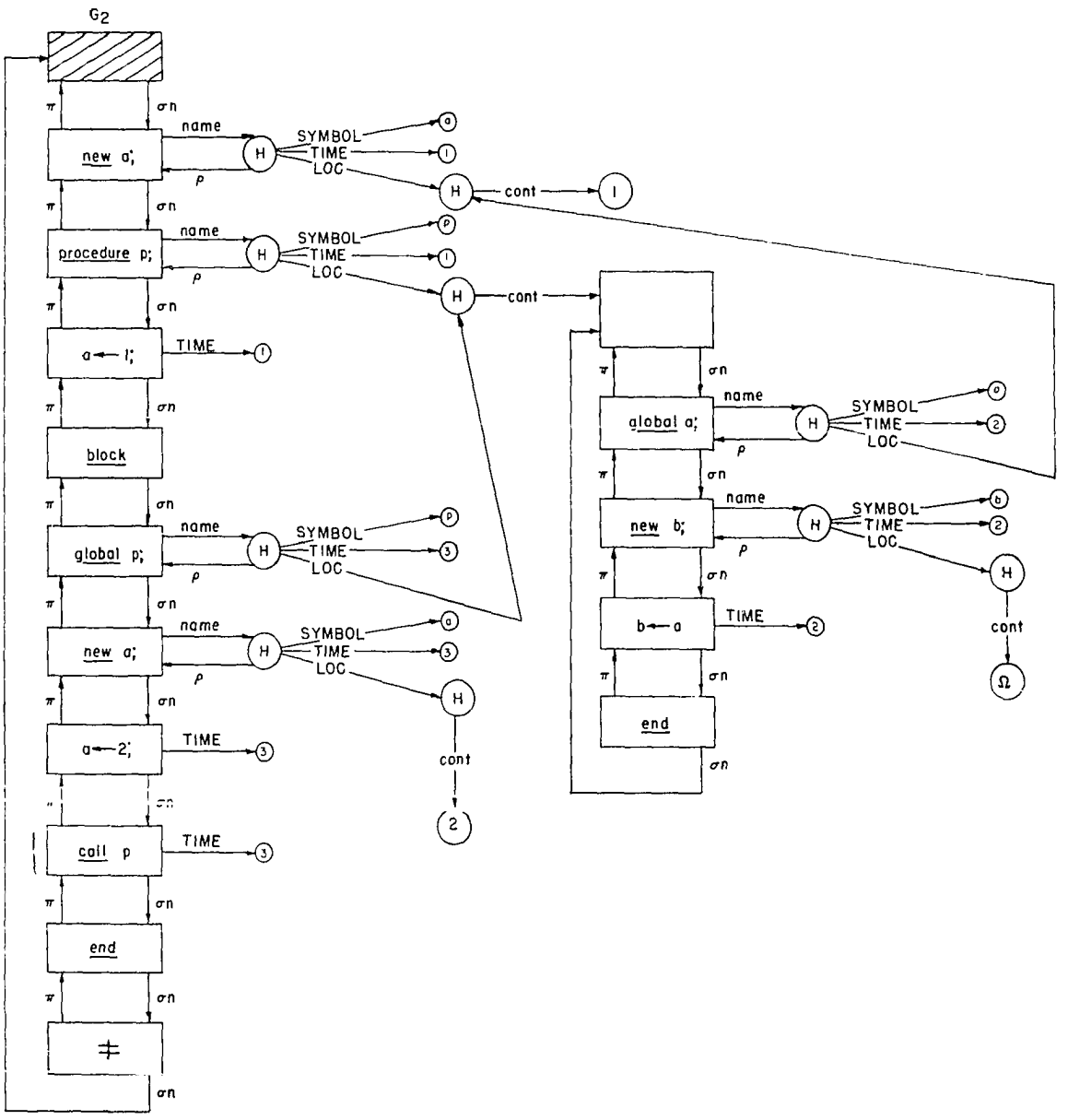


Figure 18b. Execution trace for example 12 with state = $(G_2, \sigma n^8)$

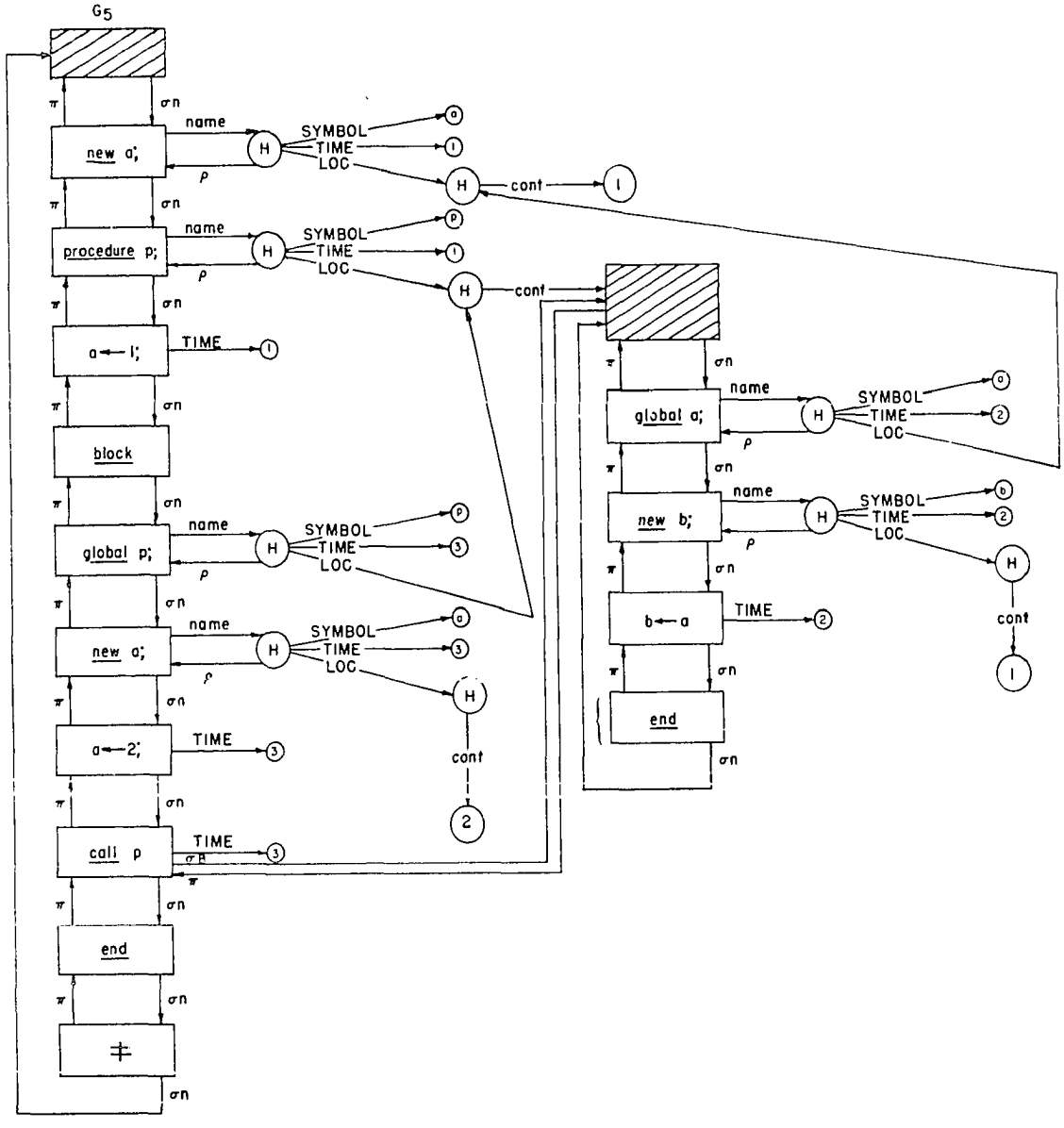


Figure 18c. Execution trace for example 12 with state = $(G_5, \sigma n^8 \sigma B \sigma^4)$

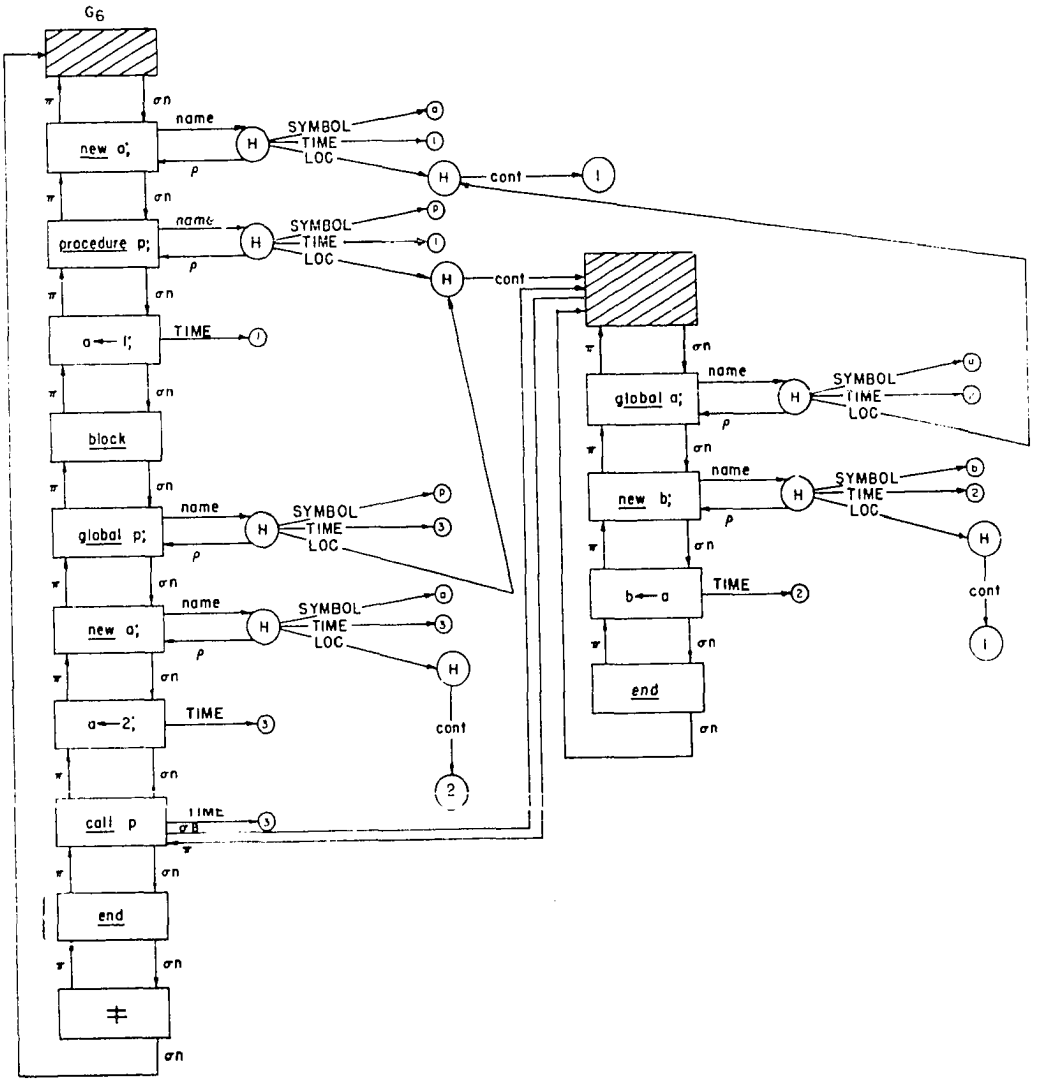


Figure 18d. Execution trace for example 12 with state = $(G_6, \sigma n^9)$

blocks. Recursion is not allowed in SPL and, in fact, all calls on a given procedure use the same local names. Since all names are retained in SPL, the values of the 'cont' properties of local subnames are retained between calls.

In the AGM, a procedure call is accommodated by linking a 'σB' property at the point of call. The value of this property is the blank header node for the procedure's text.

Example 13

SPL Program

```
new a;
procedure q;
    new c;
    c ← 5;
    return (c)
end;
a ← q
ǂ
```

A partial execution trace of the program listed in Example 13 is given in Figures 19a-19c.

Procedures in SPL may have the ability to return values. Whether or not they possess this capability is determined at translation time. When it is determined that a procedure possesses this capability, a special return name with 'SYMBOL' property "$r" is created as a 'name' property value of the blank header node for the procedure. At translation time, the 'SYMBOL' property value and the 'TIME' property value are established for this name. The determination of the 'loc' and 'cont' property values must be deferred until execution time since different

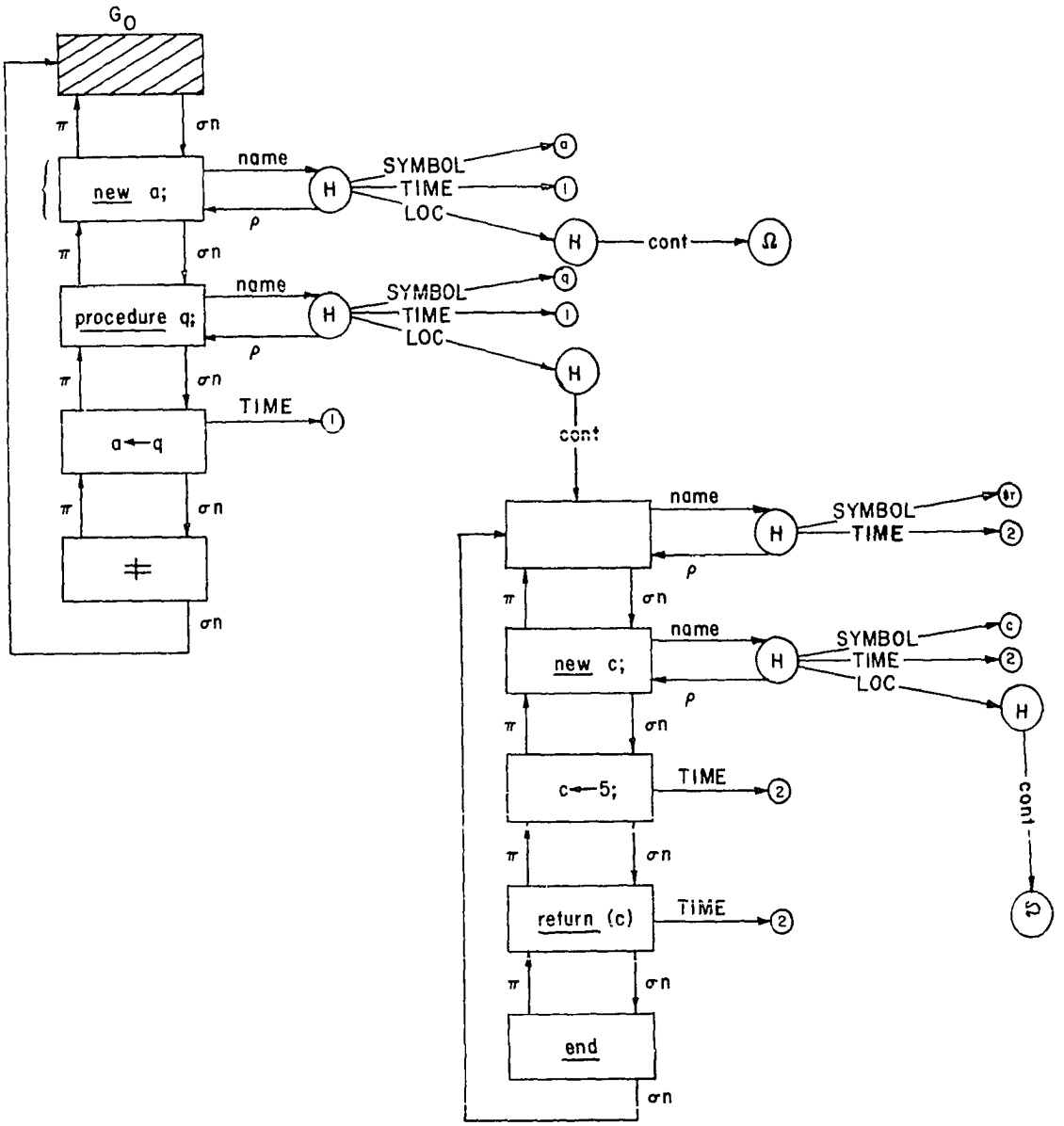


Figure 19a. Initial AGM representation for example 13 with state = $(G_0, \sigma n)$

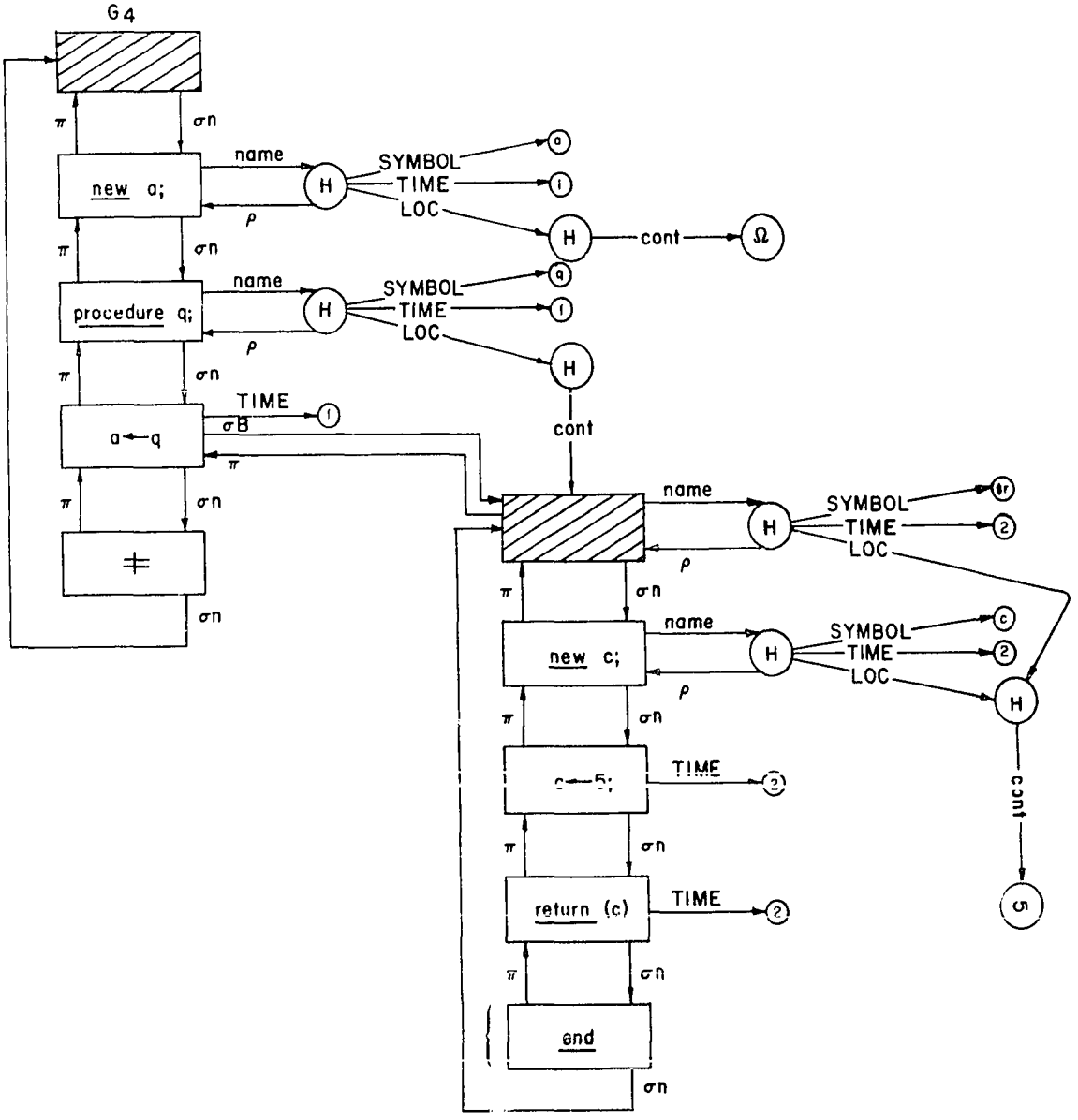


Figure 19b. Execution trace for example 13 with state = $(G_4, \sigma n^3 \sigma B \sigma n^4)$

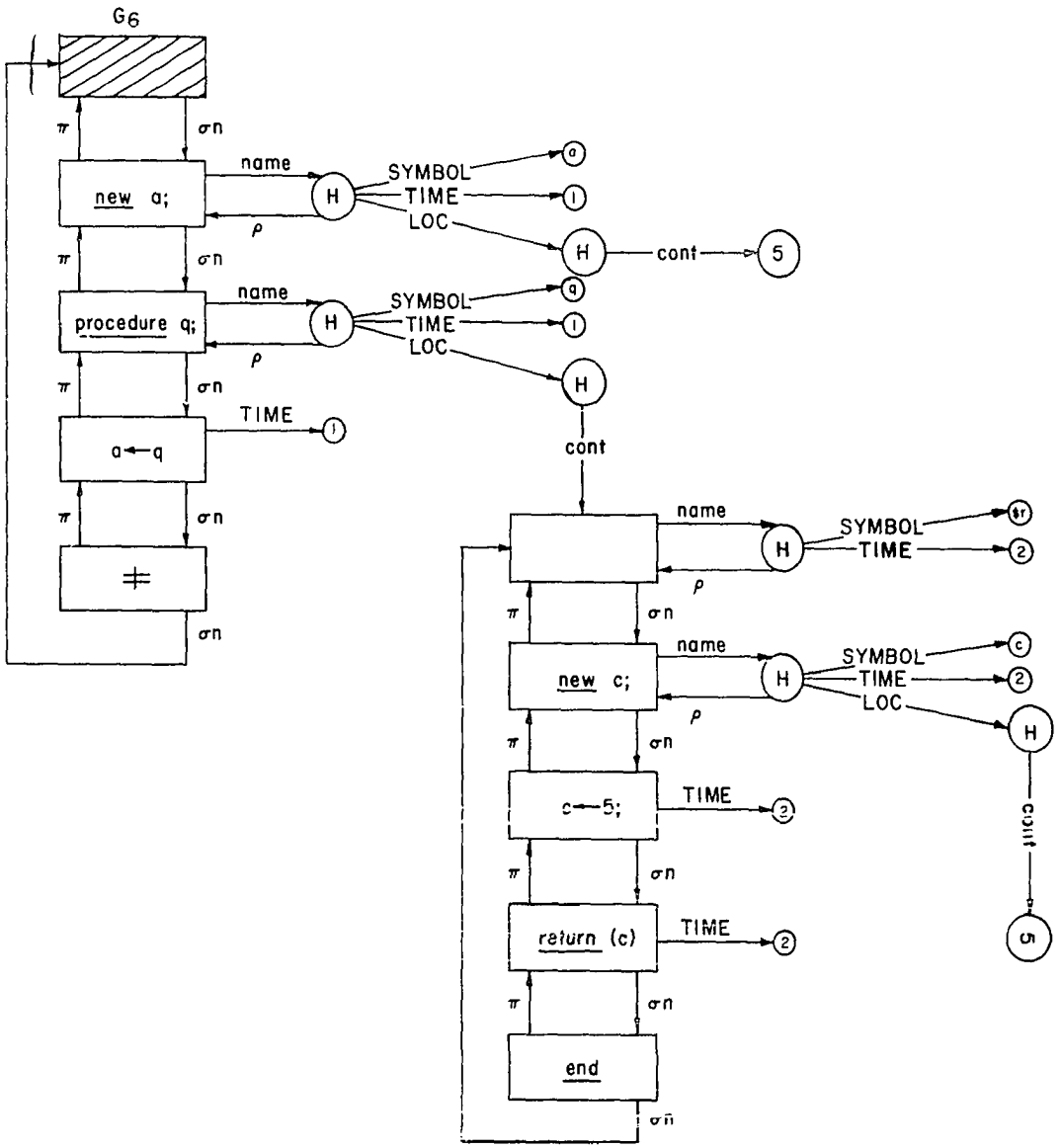


Figure 19c. Execution trace for example 13 with state = $(G_6, \sigma n^5)$

actions are required for returning expressions (e.g., "a+b") and for returning simple names (e.g., "a"). In the case of expressions, the header node for the return name acquires its own 'loc' property value where the value of 'cont' property is the calculated value of the expression. For simple names, the value of the 'loc' property of the header node for the return name is the subname (the 'loc' property value) of the simple name.

Function-type procedure calls may appear in either a left-hand or right-hand context. In exiting a procedure via a return, the 'loc' property value of the return name is passed back if a simple name is being returned. If an expression is the return argument then the 'loc'.'cont' property of the return name is passed back. Only procedures that return simple names can be used in left-hand context.

Example 14

SPL Program

```
new a;
new b;
procedure p(x,y);
    new z;
    x ← 4;
    z ← x+y
end;
a ← 1;
b ← 2;
call p(a,a+b)
ǂ
```

A partial execution trace of the program listed in Example 14 is given in Figures 20a-20b.

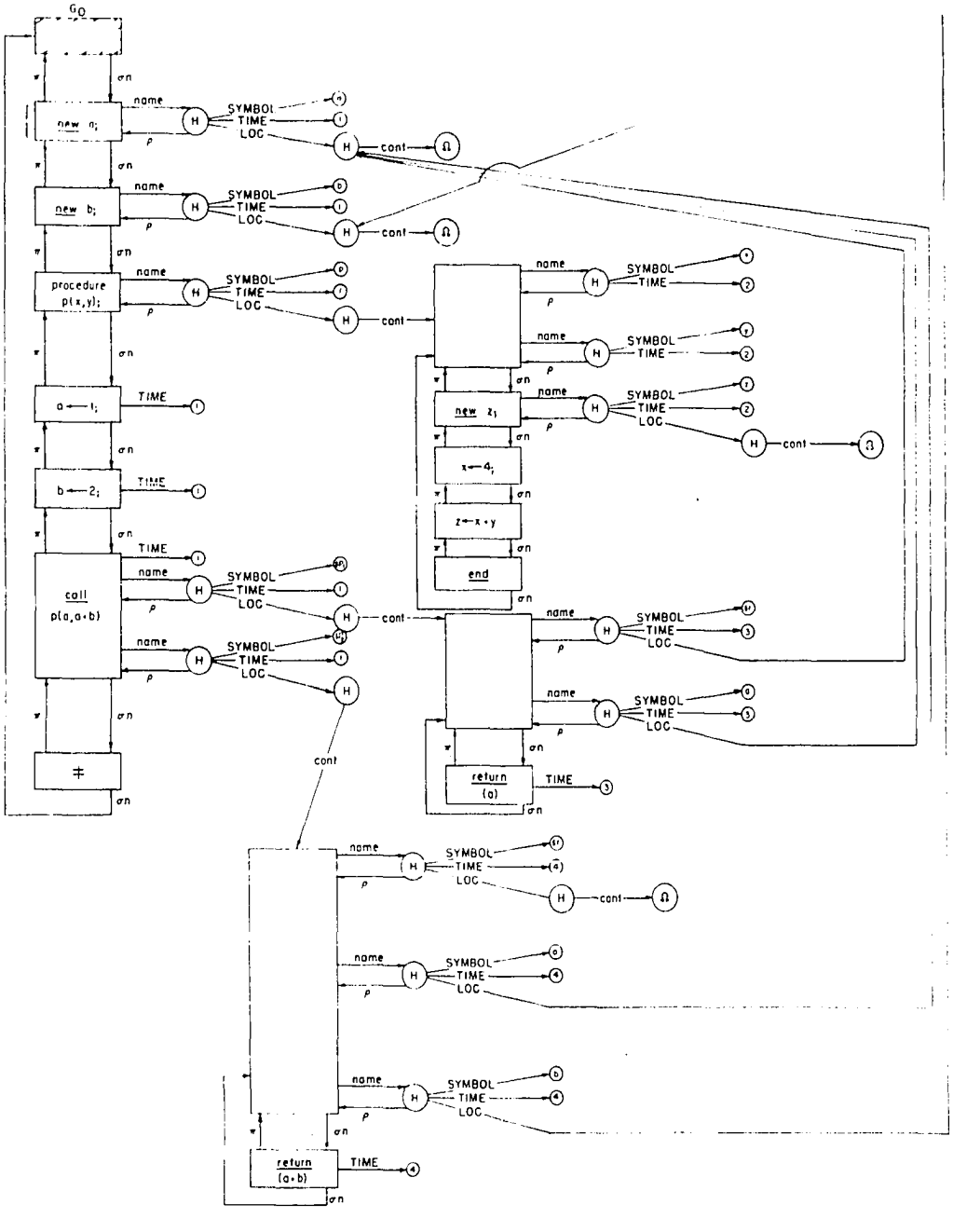


Figure 20a. Initial AGM representation for example 14 with state = (G_0, σ)

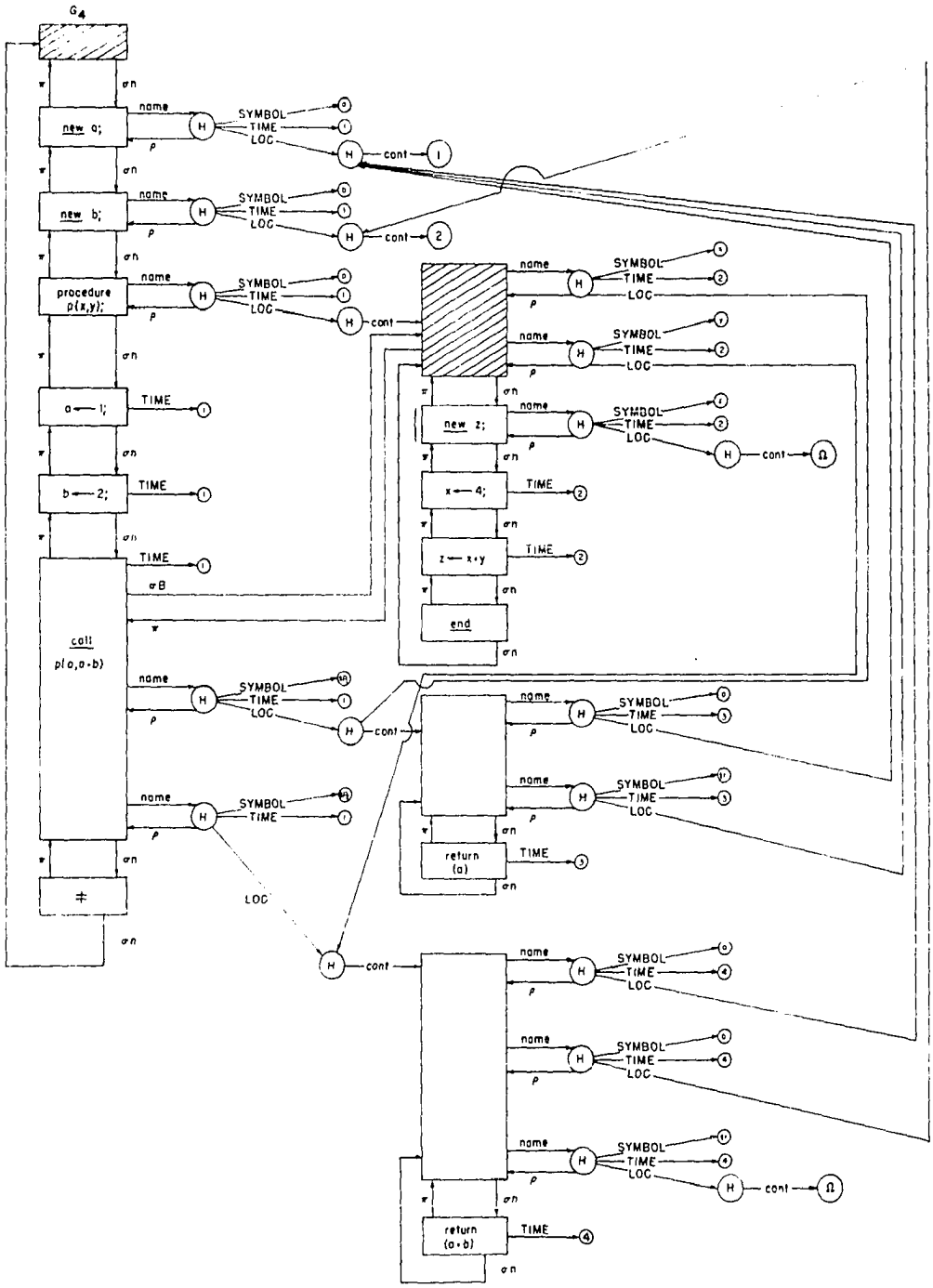


Figure 20b. Execution trace for example 14 with state = $(G_4, \sigma n^6 \sigma B \sigma n)$

The default form of parameter passage in SPL is call by name. Modeling the call by name feature in the AGM is accomplished by establishing function type procedures for the actual parameters. At translation time, when a procedure with parameters is declared, a name is created for each formal parameter. These names are associated with the blank header node of the 'loc'.'cont' value of the procedure's name. The 'SYMBOL' property values of these names are specified to be the formal parameter identifiers and the 'TIME' property values are the value of the system clock at the time of procedure translation. The 'loc' properties of these names are left undefined at translation time.

When a procedure call is translated (for procedures with parameters), a name is created for each actual parameter. These names are associated with the program node containing the call. These names are assigned 'SYMBOL' property values $\$p_1$, $\$p_2$....$\$p_n$, where $pi denotes the 'SYMBOL' value of the i^{th} actual parameter 'name' property. The 'TIME' property values of these names are the value of the system clock at the time of the translation of the call. The 'loc' properties of these names have a subname value, where the 'cont' property of the subname is a function type procedure. The procedure corresponding to the 'name'.'loc'.'cont' value of the i^{th} actual parameter is a program node containing the code "return (Ai)", where Ai is the i^{th} actual. The blank header nodes for these procedures have name values for $r, the return name, and for each identifier appearing in the actual. In the latter case, the created names are treated as global declarations. Since the return value code is determined for these procedures, the 'loc' and 'cont' property values

are completely specified.

The linking of formal parameters to their actual parameter counterparts is performed at execution time when the call is executed. At this time the 'loc' property of the formal is specified to be the same as the 'loc' property of the corresponding actual. The calling node acquires a σB property whose value is the blank header node of the called procedure and a dual π property is defined. Referencing a formal in the procedure results in a call on the actual's procedure.

In attempting to execute "x ← 4" in procedure p, the usual NAL assignment code is

attachprop (propval (propx (propset (N, 'TIME', propval (σn^6 σB σn^2, 'TIME')), 'SYMBOL', x), 'loc'), {('cont', 4)}); .

Since the subname value of x has a 'cont' value which is a procedure, the procedure is executed and the 'loc' value of the return name is used in the update. The end result is that "a" acquires a 'loc'.'cont' value of 4. Similarly, "z ← x+y" results in "z" acquiring a 'loc'.'cont' value of 6.

Example 15

SPL Program

```
new z;
new x;
new y;
new a;
new b;
a ← 1;
b ← 2;
x ← link a;
y ← link a+b;
z ← x+y
‡
```

A partial execution trace for the program listed in Example 15 is given in Figures 21a-21b.

SPL supports a generalized type of assignment called the link assignment. The form of this assignment is

"< link-variable> ← link < link-expression> ".

The effect of the assignment is to bind the link expression (as code) to the link variable. After execution of a link assignment, a reference occurrence of the link variable results in execution of the link expression bound to it. This execution takes place in the environment in which the link is created. In this sense, evaluation of a link variable is similar to evaluation of a 'call by name' formal parameter.

In the AGM, the code which appears as a link expression is transformed to a function type procedure just as for actual parameters. A return name is established and for each identifier appearing in the link expression (procedure body) a name is created as for actual parameters.

At execution time, the execution of a statement of the form

"< link-variable> ← link < link-expression> "

results in the specification of the 'loc' property of the link variable to be the 'loc' property of the "< link-expression> " name. Subsequent reference to a link variable then results in the execution of the link expression code. Thus, in the program of Example 15, execution of the program node containing "x ← 4" results in the updating of the 'loc'.'cont' property value of 2 to 4. The 'loc'.'cont' property value of z is subsequently set to 6.

As can be noted by examining the graphs for Examples 14 and 15, the

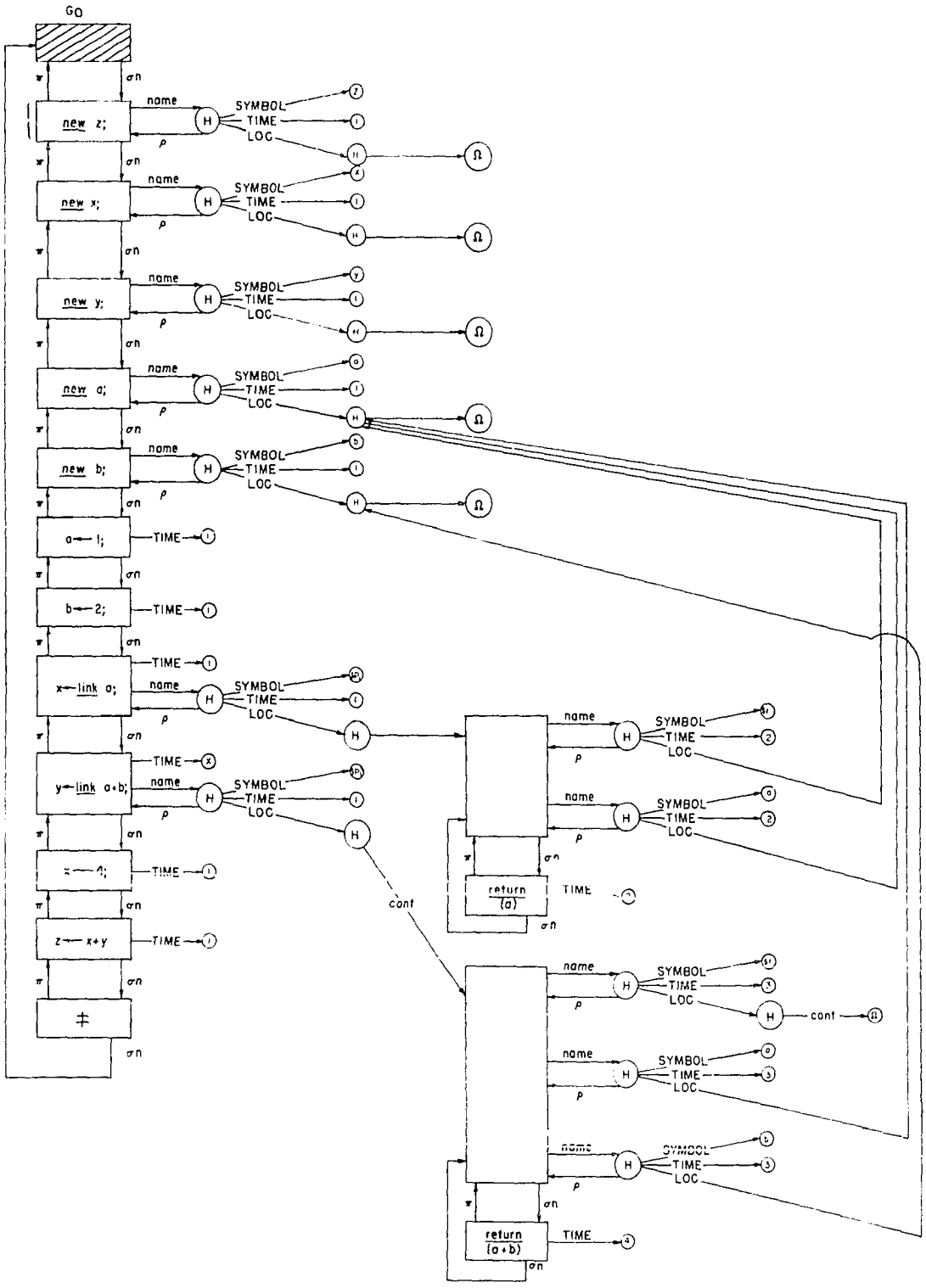


Figure 21a. Initial AGM representation for example 15 with state = (G_0, σn)

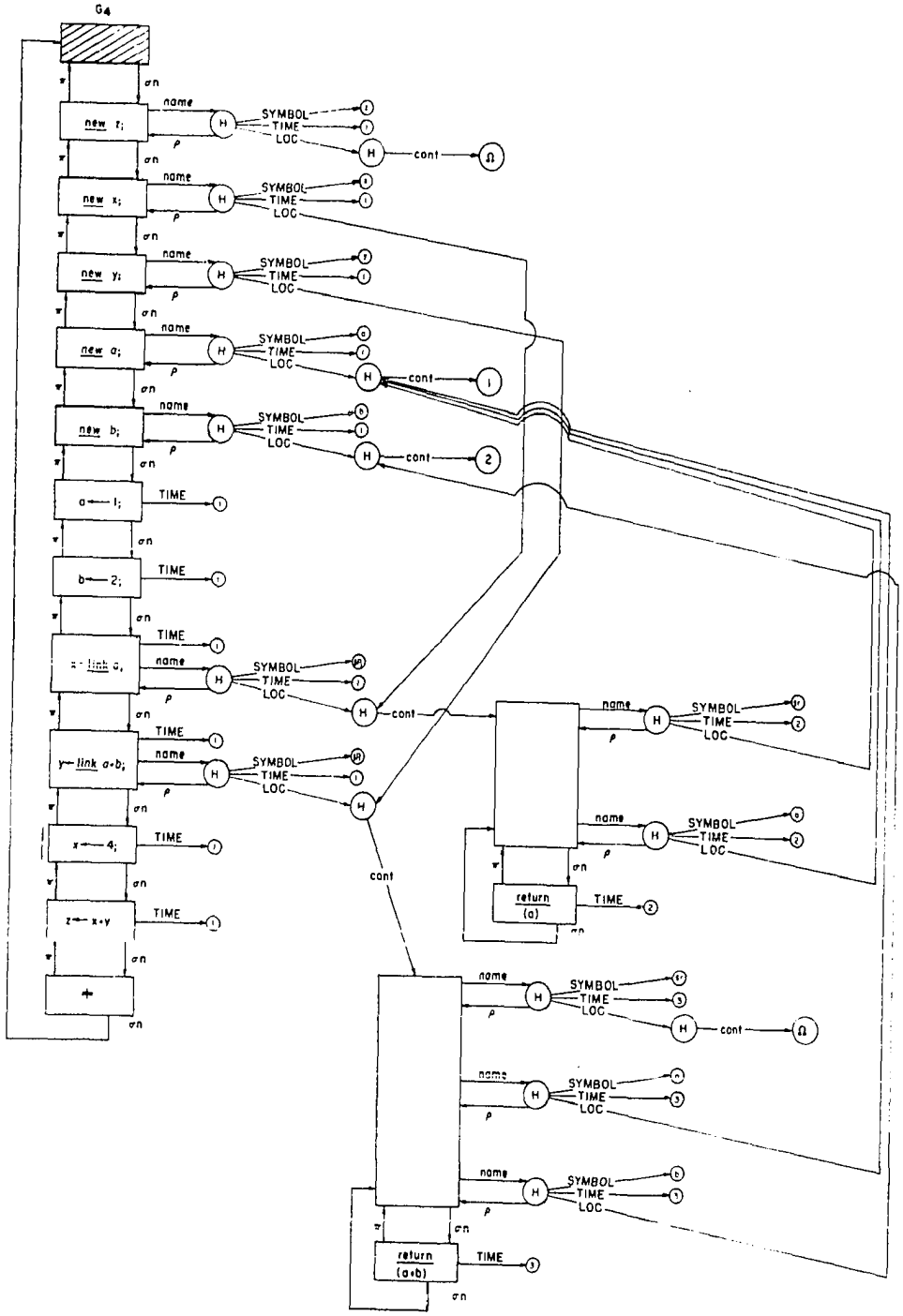


Figure 21b. Execution trace for example 15 with state = $(G_4, \sigma n^{10})$

link can provide a more economical realization of programs containing "short" procedures with parameters (1 or 2 lines). The effect of call by name can be achieved with the link. Figures 22 and 23 demonstrate how the link construct can be used to simulate nonparameterized and parameterized, single statement, function type procedures.

```
{ TEXT 1
block
global G1,...,GJ;
new N1,...,NI;
procedure Q;
    global g1,...,gK;
    return (exp)
end;
{ TEXT 2
Q;
{ TEXT 3
}
```

```
{ TEXT 1
block
global G1,...,GJ;
new N1,...,NI;
new Q;
Q ← link exp;
{ TEXT 2
Q;
{ TEXT 3
}
```

Figure 22. Link simulation of a nonparameterized procedure

```
{ TEXT 1
block
global G1,...,GJ:
new N1,...,NI;
procedure Q(F1,...,FK);
    global g1,...,gM;
    return (exp)
end;
{ TEXT 2
Q (B1,...,BK);
{ TEXT 3
≠
```

```
{ TEXT 1
block
global G1,...,GJ;
new N1,...,NI;
new Q,Fi,...,FK;
Q ← link exp;
{ TEXT 2
F1 ← link A1;
   .
   .
   .
FK ← link AK;
Q;
{ TEXT 3
F1 ← link B1;
   .
   .
   .
FK ← link BK;
Q;
{ TEXT 4
≠
```

Figure 23. Link simulation of a parameterized procedure

CHAPTER VI. CONCLUSIONS AND FUTURE RESEARCH

The name accessing problem in programming languages is both important and difficult. The work presented in this dissertation represents only a modest first step in attacking the problem. In this regard, the principal contributions of this work are listed below.

1) The description of a name in terms of a set of (P,V) pairs seems to be quite useful for discussing name accessing. In most semantic models, name accessing is discussed in terms of a set of name accesing functions, i.e., f_1, f_2,.... Each f_i normally takes one argument, an identifier, and returns either that identifier's value or its location. Thus, a name is viewed simply as an identifier. The need for a set of name accessing functions arises because an identifier may have different meanings at different points in a computation. A change in an identifier's meaning is accommodated not by a change in the properties associated with the identifier, but by a change in the name accessing function used to access the identifier.

   In generalizing the notion of a name, i.e., using the (P,V) pair concept, only one name accessing function is needed. This function has the form

$$f(N, \{(P_1,V_1),\ldots,(P_n,V_n)\}),$$

   where N is the universe of names and each (P_i,V_i) pair denotes a (property, value) pair used in selecting a given name. The unification provided by this single name accessing function appears to

offer dividends with respect to analyzing name accessing alternatives within a given language. For instance, in discussing ML-3, it was noted that the name accessing function could be applied in two distinct ways depending on the order in which the (P,V) pairs were applied in selecting elements from N. The single name accessing function approach also seems to offer a systematic approach toward comparing name accessing in different programming languages.

2) The notion that certain properties of a name must remain fixed throughout a computation in order to guarantee unique accessing offers the hope that programming languages can be classified in terms of their name accessing mechanisms. The identification of such fixed (i.e., intrinsic) properties represents a first step in the identification of the invariants in programs.

3) The specification of a set of primitive operations for discussing the name accessing problem represents a set of postulates regarding the naming process itself. The set of primitive operations specified in this dissertation seems to work well for a wide class of naming constructs. They also seem adequate for accommodating many of the naming mechanisms in the SYMBOL-2R Programming Language. No claim is made, however, that these primitives are either necessary or sufficient to accommodate naming mechanisms in all programming languages.

4) In the author's opinion, the introduction of the Accessing Graph Model provides a powerful descriptive device for the modeling of name accessing mechanisms. The use of access paths to model relationships between various computational entities could be useful in

illustrating and proving the existence of controlled sharing and protection.

5) The machinery developed in this dissertation represents a framework within which computer scientists can examine the name accessing problem. It is shown here that this framework can accommodate many naming issues. The ultimate test of this framework, however, will be its usefulness in accommodating real problems. The outcome of such a test will be the object of future study.

From the outset, it was hoped that the work reported in this dissertation would serve as a foundation for additional research. In the sense that this dissertation raises more questions than it answers, this hope has been realized. Some questions deserving particular attention in the near future are listed below.

1) Does there exist a relationship between the number of intrinsic properties associated with names in a particular language and the inherent complexity of the name accessing problem for that language? In fact, what is meant by the inherent complexity of the name accessing problem for a language? If this term is precisely defined, could it be used as a framework for comparing languages on the basis of this complexity?

2) Can we talk about a necessary and sufficient set of primitives for the accommodation of name accessing in programming languages? How do we know when a given set is necessary? How do we know when a given set is sufficient? Is it meaningful for a set of naming primitives

to be complete or consistent in the sense that a set of axioms is complete or consistent?

3) In what sense is the graph theoretic, access path approach used in the AGM applicable to the important issues of sharing and controlled access? Can this approach be used to discuss these issues within the context of real information systems?

4) To what extent can the model presented here be formalized? Can this model be represented as a framework within which questions can be phrased (and perhaps even answered) in a mathematically rigorous way? Is it worthwhile attempting this?

5) Can we use this model to learn more about the design and implementation of programming languages? In what sense is this model may better (or worse) than other such models?

6) In what sense is a theory of names related to a theory of types?

The questions listed above are by no means exhaustive. The realization of a general theory of names for programming languages will, however, require that these questions (and others) be addressed.

BIBLIOGRAPHY

1. Anderberg, J. W. and Smith, C. L. "High-Level Language Translation in SYMBOL-2R." Proceedings of a Symposium on High-Level-Language Computer Architecture, SIGPLAN Notices, 8 (November, 1973), 11-19.

2. Bakker, J. W. de "Formal Definition of Programming Languages." Mathematical Center Tracts 16. Amsterdam: Mathematisch Centrum, 1967.

3. Bakker, J. W. de "Semantics of Programming Languages." Advances in Information Systems Science 2. Edited by J. T. Tou. New York: Plenum Press, 1969, 173-228.

4. Bekic, H.; Bjorner, D.; Henhapl, W.; Jones, C. B.; and Lucas, P. "A Formal Definition of a PL/I Subset." Technical Report TR25.139, IBM Laboratory Vienna, 1974.

5. Berry, D. M. "Introduction to Oregano." Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices, 6 (February, 1971), 171-190.

6. Boyle, J. M. and Grau, A. A. "An Algorithmic Semantics for ALGOL 60 Identifier Denotation." Journal of the ACM, 17 (April, 1970), 361-382.

7. Cadiou, J. M. and Manna, Z. "Recursive Definitions of Partial Functions and their Computations." Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, 7 (January, 1972), 58-65.

8. Caracciolo, A. "On the Concept of Formal Linguistic Systems." Formal Language Description Languages for Computer Programming. Edited by T. B. Steel, Jr. Amsterdam: North-Holland Publishing Company, 1966, 37-51.

9. Caracciolo, A. and Wolkenstein, N. "On a Class of Programming Languages for Symbol Manipulation Based on Extended Markov Algorithms." Report No. 21, Pisa: Centro Studi Calcolatrici Electtraniche, 1963.

10. Chesley, G. and Smith, W. "The Hardware-implemented High-Level Machine Language for SYMBOL." Proceedings of the Spring Joint Computer Conference 38. Montvale, New Jersey: AFIPS Press, 1971, 563-573.

11. Church, A. "The Calculi of Lambda-Conversion." Annuals of Mathematical Studies 6. Princeton, New Jersey: Princeton University Press, 1951.

12. Dennis, J. B. "On the Design and Specification of a Common Base Language." Proceedings of the Symposium on Computers and Automation. Brooklyn, New York: Polytechnic Press of the Polytechnic Institute of Brooklyn, 1971, 47-74.

13. Dijkstra, E. W. "Recursive Programming." Programming Systems and Languages. Edited by S. Rosen. New York: McGraw-Hill Book Company, 1967, 221-227.

14. Earley, J. "Toward an Understanding of Data Structures." Communications of the ACM, 14 (October, 1971), 617-627.

15. Earley, J. "Relational Level Data Structures in Programming Languages." Acta Informatica, 2 (1973), 293-309.

16. Earley, J. and Caizerques, P. "VERS Manual." Computer Science Department, University of California, Berkeley, 1971.

17. Ellis, D. J. "Semantics of Data Structures and References." Computation Structures Group Memo 106, Project MAC, Massachusetts Institute of Technology, 1974.

18. Elson, M. Concepts of Programming Languages. Chicago: Science Research Associates, Inc., 1973.

19. Fang, I. "FOLDS, A Declarative Formal Language Definition System." Computer Science Report CS-72-329, Computer Science Department, Stanford University, 1972.

20. Feldman, J. A. "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler." Communications of the ACM, 9 (January, 1966), 3-9.

21. Floyd, R. W. "Assigning Meanings to Programs." Mathematical Aspects of Computer Science 19. Edited by J. T. Schwartz. Providence, Rhode Island: American Mathematical Society, 1967, 19-32.

22. Garwick, J. V. "The Definition of Programming Languages by their Compilers." Formal Language Description Languages for Computer Programming. Edited by T. B. Steel, Jr. Amsterdam: North Holland Publishing Company, 1966, 139-147.

23. George, J. and Sager, G. "Variables - Bindings and Protection." SIGPLAN Notices, 8 (December, 1973), 18-29.

24. Gilmore, P. C. "An Abstract Computer with a Lisp-Like Machine Language without a Label Operator." Computer Programming and Formal Systems. Edited by P. Braffort and D. Hirschberg. Amsterdam: North Holland Publishing Company, 1967, 71-86.

25. Goguen, J. A. and Thatcher, J. W. "Initial Algebra Semantics." Proceedings of the 15th IEEE Conference on Switching and Automata Theory, 15 (1974), 63-77.

26. Goguen, J. A.; Thatcher, J. W.; Wagner, E. G.; and Wright, J. B. "A Junction between Computer Science and Category Theory, I: Basic Concepts and Examples (Part 1)." IBM Technical Report RC4526, IBM Thomas J. Watson Research Center, September, 1973.

27. Henhapl, W. and Jones, C. B. "The Block Structure Concept and Some Possible Implementations with Proofs of Equivalence." Technical Report TR25.104, IBM Laboratory Vienna, 1970.

28. Higman, B. A Comparative Study of Programming Languages. New York: American Elsevier Publishing Company, Inc., 1967.

29. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming." Communications of the ACM, 12 (October, 1969), 576-580, 583.

30. Hoare, C. A. R. and Lauer, P. E. "Consistent and Complementary Formal Theories of the Semantics of Programming Languages." Acta Informatica, 3 (1974), 135-153.

31. Hoare, C. A. R. and Wirth, N. "An Axiomatic Definition of the Programming Language PASCAL." Acta Informatica, 2 (1973), 335-355.

32. Hutchison, P. C. and Ethington, K. "Program Execution in the SYMBOL-2R Computer." Proceedings of a Symposium on High-Level-Language Computer Architecture, SIGPLAN Notices, 8 (November, 1973), 20-26.

33. IFIP, "Report on Subset ALGOL 60 (IFIP)." Communications of the ACM, 7 (October, 1964), 626-628.

34. Ingerman, P. Z. "Dynamic Declarations." Communications of the ACM, 4 (January, 1961), 59-60.

35. Johnston, J. B. "The Contour Model of Block Structured Processes." Proceedings of a Symposium on Data Structures in Programming Languages, SIGPLAN Notices, 6 (February, 1971), 55-82.

36. Johnston, J. B. "Identifier Binding and Access in Nested Declaration Computations." Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, 7 (1973), 306-312.

37. Johnston, J. B. "Binding and Flow of Control in Nested Declaration Computations." Unpublished lecture notes, Computer Science Department, New Mexico State University, 1973.

38. Johnston, J. B.; Berry, D. M.; and Murphy, D. P. "Expression Stack Management in Nested Declaration Computations." Unpublished paper, Computer Science Department, New Mexico State University, 1974.

39. Knuth, D. E. "Semantics of Context-Free Languages." Mathematical Systems Theory, 2 (June, 1968), 127-145.

40. Landin, P. J. "The Mechanical Evaluation of Expressions." The Computer Journal, 6 (January, 1964), 308-320.

41. Landin, P. J. "The Next 700 Programming Languages." Communications of the ACM, 9 (March, 1966), 157-166.

42. Ledgard, H. F. "A Formal System for Defining the Syntax and Semantics of Computer Languages." Technical Report MAC-TR60, Project MAC, Massachusetts Institute of Technology, 1969.

43. Ledgard, H. F. "Production Systems: Or Can We Do Better Than BNF?" Communications of the ACM, 17 (February, 1974), 94-102.

44. Lucas, P. "Two Constructive Realizations of the Block Concept and Their Equivalence." Technical Report TR25.085, IBM Laboratory Vienna, 1968.

45. Lucas, P.; Lauer, P.; and Stigleitner, H. "Method and Notation for the Formal Definition of Programming Languages." Technical Report TR25.087, IBM Laboratory Vienna, 1968.

46. Lucas, P. and Walk, K. "On the Formal Description of PL/I." Annual Review in Automatic Programming 6, Part 3 (1969), 105-182.

47. McCarthy, J. "Towards a Mathematical Science of Computation." Information Processing 1962. Edited by C. M. Popplewell. Amsterdam: North-Holland Publishing Company, 1963, 21-28.

48. McCarthy, J. "A Basis for a Mathematical Theory of Computation." Computer Programming and Formal Systems. Edited by P. Braffort and D. Hirschberg. Amsterdam: North-Holland Publishing Company, 1963, 33-69.

49. McCarthy, J. "A Formal Description of a Subset of ALGOL." Formal Language Description Languages for Computer Programming. Edited by T. B. Steel, Jr. Amsterdam: North-Holland Publishing Company, 1966, 1-12.

50. McGowan, C. "The 'most recent' Error; Its Causes and Corrections." Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, 7 (January, 1972), 191-202.

51. Manna, Z. "Properties of Program and the First-Order Predicate Calculus." Journal of the ACM, 16 (April, 1969), 244-255.

52. Manna, Z.; Ness, S.; and Viullemin, J. "Inductive Methods for Proving Properties of Programs." Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, 7 (January, 1972), 27-50.

53. Mosses, P. D. "Mathematical Semantics of ALGOL 60." Technical Monograph PRG-12, Oxford University Computing Laboratory, 1974.

54. Naur, P., et al. "Revised Report on the Algorithmic Languages ALGOL 60." Communications of the ACM, 6 (January, 1963), 1-23.

55. Pratt, T. W. Programming Languages: Design and Implementation. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1971.

56. Richards, H. "SYMBOL-2R Programming Language Reference Manual." Technical Report, Cyclone Computer Laboratory, Iowa State University, 1971.

57. Richards, J., Jr. and Wright, C. T., Jr. "Introduction to the SYMBOL-2R Programming Language." Proceedings of a Symposium on High-Level-Language Computer Architecture, SIGPLAN Notices, 8 (November, 1973), 27-33.

58. Richards, H., Jr. and Zingg, R. J. "The Logical Structure of the Memory Resource in the SYMBOL-2R Computer." Proceedings of a Symposium on High-Level-Language Computer Architecture, SIGPLAN Notices, 8 (November, 1973), 1-10.

59. Rosenberg, A. L. Data Graphs and Addressing Schemes." Journal of Computer and System Sciences, 5 (June, 1971), 193-238.

60. Rosenberg, A. L. "Addressable Data Graphs." Journal of the ACM, 19 (April, 1972, 309-340.

61. Rosenberg, A. L. "Transitions in Extendible Arrays." ACM Symposium on Principles of Programming Languages, 1973, 218-225.

62. Rosenberg, A. L. "Managing Storage for Extendible Arrays." Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, 6 October, 1974), 297-301.

63. Rosenberg, A. L. "Allocating Storage for Extendible Arrays." Journal of the ACM, 21 (October, 1974), 652-670.

64. Scott, D. "An Outline of a Mathematical Theory of Computation." Proceedings of the Fourth Annual Princeton Conference on Information Sciences and Systems, 4 (1970), 169-176.

65. Scott, D. and Strachey, C. "Toward a Mathematical Semantics for Computer Languages." Proceedings of the Symposium on Computers and Automata. Brooklyn, New York: Polytechnic Institute of Brooklyn, 1971, 19-46.

66. Steel, T. B., Jr. "A Formalization of Semantics of Programming Language Description." Formal Language Description Languages for Computer Programming. Edited by T. B. Steel, Jr. Amsterdam: North-Holland Publishing Company, 1966, 25-36.

67. Strachey, C. "Towards a Formal Semantics." Formal Language Description Languages for Computer Programming. Edited by T. B. Steel, Jr. Amsterdam: North-Holland Publishing Company, 1966, 198-220.

68. Strachey, C. "The Varieties of Programming Languages." Technical Monograph PRG-10, Oxford University Computing Laboratory, 1973.

69. Tarski, A. "A Lattice-Theoretical Fixpoint Theorem and Its Applications." Pacific Journal of Mathematics, 15 (May, 1955), 285-309.

70. Tennent, R. D. "Mathematical Semantics and Design of Programming Languages." Technical Report No. CSRG-39, Computer Systems Research Group, University of Toronto, 1973.

71. van Wijngaarden, A. "Recursive Definition of Syntax and Semantics." Formal Language Description Languages for Computer Programming. Edited by T. B. Steel, Jr. Amsterdam: North-Holland Publishing Company, 1966, 13-24.

72. van Wijngaarden, A.; Maulloux, B. J.; Peck, J. E. L.; and Koster, C. H. A. "Report on the Algorithmic Language ALGOL 68." Numerische Mathematic 14. New York: Springer-Verlag, 1969, 79-218.

73. Walk, K.; Alber, K.; Bandat, K.; Bekic, H.; Chroust, G.; Kudielka, V.; Oliva, P.; and Zeisel, G. "Abstract Syntax and Interpretation of PL/I." Technical Report TR25.082, IBM Laboratory Vienna, 1968.

74. Wegbreit, B. "Procedure Closure in EL1." The Computer Journal, 16 (February, 1974), 38-43.

75. Wegner, P. "The Variability of Computations." Technical Report No. 70-22, Center for Computing and Information Sciences, Brown University, 1970.

76. Wegner, P. "Programming Language Semantics." Formal Semantics of Programming Languages. Edited by R. Rustin. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1970, 149-248.

77. Wegner, P. "The Vienna Definition Language." ACM Computing Surveys, 4 (March, 1972), 5-63.

78. Wilner, W. T. "Formal Semantic Definition Using Synthesized and Inherited Attributes." Formal Semantics of Programming Languages. Edited by R. Rustin. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1970, 25-40.

79. Wirth, N. and Weber, H. "EULER: A Generalization of ALGOL, and its Formal Definition: Part I." Communications of the ACM, 9 (January, 1966), 13-23, 25.

80. Wirth, N. and Weber, H. "EULER: A Generalization of ALGOL, and its Formal Definition: Part II." Communications of the ACM, 9 (February, 1966), 89-99.

ACKNOWLEDGMENTS

The support that was received during the research and writing of this dissertation has ranged from technical to moral and all of it has been important. The following summarizes the various roles played in the production of this dissertation.

Directors	Robert M. Stewart Charles T. Wright, Jr.
Technical Assistance Coordinator	Charles T. Wright, Jr.
Typist (final version)	Gwen Ethington
Typists (initial versions)	Wanda Lembke Betty Dahlgren Debbie Hagebock
Thou-Shalt-Not-Quit Chorus	"The Family" Patricia Bulger Betty Dahlgren Karen Ethington Robert M. Stewart Charles T. Wright, Jr.
Private Fan Club	Christine L. Smith Kelly L. Smith Paul R. Smith
Associate Producers	John J. Murphy Patrick J. Murphy